Maisterintutkielma

Tietojenkäsittelytiede

Jatkuvan kokeilemisen tekninen toteutus

Simo Kolppo

4.5.2020

Matemaattis-luonnontieteellinen tiedekunta

Helsingin yliopisto

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähkopostiosoite: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytiede

Tekijä — Författare — Author

Simo Kolppo

Työn nimi — Arbetets titel — Title

Jatkuvan kokeilemisen tekninen toteutus

Ohjaajat — Handledare — Supervisors

yliopistotutkija Timo Asikainen, professori Tomi Männistö

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Maisterintutkielma	4.5.2020	50 sivua, 51 liitesivua

Tiivistelmä — Referat — Abstract

Jatkuva kokeileminen tarkoittaa ohjelmistoon tehtävien kokeiluiden keskeistä asemaa ohjelmistokehityksessä. Näillä kokeiluilla pyritään selvittämään uudesta toiminnallisuudesta saatavaa hyötyä tai haittaa ennen kuin ne liitetään pysyvästi osaksi ohjelmistoa.

Työn tarkoitus on tarkastella, mitä jatkuvan kokeilemisen käyttöönotto vaatii sekä jatkuvan kokeilemisen teknisen toteutuksen kannalta olennaisia osa-alueita. Osa-alueiksi työhön valittiin testi- ja kontrolliversioiden variaatioiden toteutus, datan keräys, käsittely ja analysointi sekä kokeilun kohteen valinta.

Jatkuvan kokeilemisen käyttöönoton haasteiden ja keskeisten osa-alueiden analysoiminen tehdään tässä työssä kirjallisuuskatsauksena. Ensin työssä taustoitetaan jatkuvan kokeilemisen kannalta keskeisiä konsepteja: A/B-testausta, jatkuvaa integraatiota ja jatkuvaa toimitusta. Tämän jälkeen tutkitaan valituista jatkuvan kokeilemisen osa-alueista useita erilaisia lähestymistapoja.

Tutkielmasta ilmenee, että monien osa-alueiden toteutukseen ei ole yhtä parasta toteutustapaa, vaan erilaiset ohjelmistot ja organisaatiot vaativat erilaisia ratkaisuja. Usein näillä ratkaisuilla on kullakin omat hyvät ja huonot puolensa, jotka on syytä ottaa huomioon. Tutkielmasta ilmenee kuitenkin monia yleisesti hyviä käytäntöjä jatkuvan kokeilemisen toteutukseen.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management → Software development techniques
Human-centered computing → Human computer interaction (HCI) → HCI design and evaluation methods → User studies

Avainsanat — Nyckelord — Keywords

jatkuva kokeileminen, A/B-testaus, ohjelmistokehitys

Säilytyspaikka — Förvaringsställe — Where deposited

Helsingin yliopiston kirjasto

Muita tietoja — övriga uppgifter — Additional information

Ohjelmistojärjestelmien erikoistumislinja

Sisältö

1 Johdanto **1**

1.1 Työn tausta ja motivaatio 1

1.2 Työn tavoitteet ja tutkimuskysymykset 2

1.3 Menetelmät 3

1.4 Työn rakenne 3

2 A/B-testaus **5**

2.1 Kokonaisarviointikriteerit 7

3 Jatkuva integraatio ja jatkuva toimitus **9**

3.1 Jatkuva integraatio 9

3.2 Jatkuva toimitus 10

4 Jatkuva kokeileminen **13**

4.1 Siirtyminen jatkuvaan kokeilemiseen 13

4.1.1 Liiketoiminta- ja organisaatiotason vaatimukset 14

4.1.2 Tekniset vaatimukset 16

4.2 Testi- ja kontrolliversioiden variaatioiden toteutus 19

4.2.1 Ominaisuuskytkimet 19

4.2.2 Ajonaikainen liikenteen jakaminen 21

4.2.3 Ulkopuolinen palvelu 23

4.2.4 Käyttäjien jakaminen testi- ja kontrolliryhmiin 24

4.3 Datan keräys, käsittely ja analysointi 27

4.3.1 Yleiskatsaus lokitukseen 27

4.3.2 Datan keräys jatkuvassa kokeilemisessa 30

4.3.3 Datan jatkokäsittely 33

4.3.4 Datan analysointi 35

4.3.5 Reaaliaikainen monitorointi 38

4.4 Kokeilun kohde 39

4.4.1 Kokeilujen laajuus 41

5 Yhteenveto **44**

5.1 Tarkastelu ja jatkotutkimukset 45

Kirjallisuus **47**

1 Johdanto

Jatkuvan kokeilemisen käyttö ohjelmistokehityksessä tarkoittaa erilaisten kokeiluiden käyttöä olennaisena osana ohjelmistokehitysprosessia. Kokeiluilla pyritään niissä kerätyn datan perusteella selvittämään parantaako ohjelmistoon tehty muutos sitä. Muutos otetaan käyttöön vain mikäli kerätty data osoittaa sen parantavan ohjelmistoa. Valtaosa näistä kokeiluista tehdään A/B-testeinä (Deng ja Shi, 2016), mutta myös muunlaisilla menetelmillä voidaan kerätä dataa muutoksen hyödyistä ja haitoista.

1.1 Työn tausta ja motivaatio

Laadukkaalle ohjelmistolle on jatkuvasti kasvava kysyntä, mutta millainen on laadukas ohjelmisto? Käyttäjien näkökulmasta yksi tärkeimmistä ohjelmiston ominaisuuksista on sen käytettävyys ja helppokäyttöisyys. Ohjelmistoyritykset ja -kehittäjät yrittävät luonnollisesti tarjota ohjelmistojensa käyttäjille parhaan mahdollisen käyttökokemuksen, mutta parhaan mahdollisen, tai edes hyvän, käyttökokemuksen tarjoaminen ei välttämättä ole helppoa.

Ohjelmistoprojekteissa on yleensä monta mielipidettä siitä, miten toteutetaan hyvä käyttäjäkokemus. Mahdollisesti jopa yhtä monta kuin projektissa on henkilöitä mukana. Ongelmana näissä on kuitenkin se, että ne ovat vain projektin jäsenten subjektiivisia mielipiteitä ja niiden taustalla voi olla täysin vääriä olettamuksia tai harhaluuloja käyttäjien haluista tai toimintatavoista (Kohavi, Henne et al., 2007). Tästä voi pahimmassa tapauksessa seurata se, että ohjelmistoon tehdyt muutokset vievät sitä huonompaan suuntaan ja käyttäjät alkavat etsiä vaihtoehtoisia ohjelmistoja.

Subjektiivisten mielipiteiden sijaan voidaan testata kehittäjien tai muiden ohjelmiston sidosryhmien erilaisia olettamuksia ohjelmiston käyttäjillä. Testeistä saadaan kerättyä objektiivista dataa käyttäjien käyttäytymisestä ja tämän datan pohjalta voidaan kehittää ohjelmistoa varmuudella siitä, että kehitys vie ohjelmistoa parempaan suuntaan.

Internetin, ja sen myötä selaimessa ajettavien sovellusten käytön valtava kasvu on mahdollistanut näiden käyttäjillä tehtävien kokeiluiden tekemisen ennennäkemättömällä tavalla (Kohavi, Henne et al., 2007). Monilla selainsovelluksilla on niin suuria määriä

käyttäjiä, että kokeiluita voidaan helposti kohdistaa isolle käyttäjäjoukolle hetkessä.

1.2 Työn tavoitteet ja tutkimuskysymykset

Tämän työn tavoitteena on tutkia ja kartoittaa jatkuvasta kokeilemisesta julkaistuja tieteellisiä tutkimuksia kirjallisuuskatsauksen menetelmin. Erityisesti on tarkoitus kartoittaa jatkuvan kokeilemisen teknisen toteutuksen kannalta olennaisia osa-alueita. Eri lähteiden näkökulmia näiden osa-alueiden toteutukseen vertaillaan keskenään ja niiden perusteella tehdään analyysia. Työn perustana toimii neljä tutkimuskysymystä, jotka nähtiin keskeisinä jatkuvan kokeilemisen teknisen toteutuksen kannalta.

> **Tk$_1$**: Mitä vaatimuksia tai ennakkoehtoja on jatkuvaan kokeilemiseen siirtymiseen?

A/B-testaus on yleistynyt merkittävästi viime vuosikymmenen aikana ja on monissa yrityksissä ollut arkipäivää jo vuosia (lähde). Jatkuva kokeileminen ei kuitenkaan ole läheskään yhtä laajalle levinnyt toimintatapa, vaikka tulokset osoittavat, että sillä voidaan saada merkittäviä hyötyjä ohjelmistojen kehitykseen (pari lähdettä tähän). Asettaako kokeilemisen liittäminen olennaiseksi osaksi ohjelmistokehitystä merkittäviä ennakkoehtoja organisaatiolle ja ohjelmistokehittäjille? Miten siirtyminen yksittäisistä A/B-testeistä jatkuvaan kokeilemiseen tapahtuu?

> **Tk$_2$**: Miten ohjelmisto eriytetään testi- ja kontrolliversioihin ja miten käyttäjät jaetaan näiden eri versioiden välillä?

Kun ohjelmistosta on tehty uusi testiversio, jota testata olemassa olevaa kontrolliversiota vasten, pitää siihen ohjata osa käyttäjistä. Miten varmistetaan, että suunniteltu osuus käyttäjistä ohjataan yhteen tai useampaan testiversioon ja loput kontrolliversioon? Jos kyseessä on web-sovellus, niin miten varmistetaan, että käyttäjän versio ei vaihdu latauskertojen välillä?

> **Tk$_3$**: Miten toteutetaan jatkuvan kokeilemisen datan keräys ja analysointi?

Suurimmassa osassa ohjelmistoja tehdään jonkinlaista logitusta. Tämä logitus on kuitenkin usein lähinnä testausta ja virheiden korjaamisen avuksi eikä ole riittävää käyttäjien käyttäytymisen tarkempaan analysointiin. Miten jatkuvassa kokeilemisessa toteutetaan käyttäjien käyttäytymistä mittaavan datan keräys, jatkokäsittely ja analysointi?

Tk$_4$: Miten valitaan kokeilun kohde? Onko parempi testata pieniä muutoksia vai isompia kokonaisuuksia?

Jatkuvasta kokeilemisesta on julkaistu paljon tutkimusta organisaatioiden toimintatapojen ja kulttuurin näkökulmasta (Auer ja Felderer, 2018). Tässä työssä keskitytään kuitenkin pääasiassa jatkuvan kokeilemisen teknisen toteutuksen osa-alueisiin. Poikkeuksena tähän on kappale jatkuvan kokeilemisen esiehdoista, jossa organisaatiopuolen katsottiin olevan hyvin keskeisessä asemassa.

1.3 Menetelmät

Tämä työ on tehty kirjallisuuskatsauksena analysoimalla aiheesta julkaistua ajankohtaista tieteellistä kirjallisuutta.

Työn lähdemateriaalia haettiin Google Scholar * -hakukoneesta sekä IEEExplore Digital Library † ja The ACM Digital Library ‡ -tietokannoista. Hakusanoina käytettiin ”continuous experimentation”, ”continuous experimentation architecture”, ”a/b testing”, ”controlled experiment”, ”continuous integration”sekä ”continuous delivery-termejä. Löydetyistä vertaisarvioiduista lähdeartikkeleista valittiin parhaiten aiheeseen soveltuvat ja näiden artikkeleiden lähteistä etsittiin lisää lähdemateriaalia. Lähdeartikkeiden valinnassa pyrittiin suosimaan uudempia ja ajankohtaisempia artikkeleita, joskin myös useat vanhemmat artikkelit sisältävät hyvin ajankohtaista jatkuvaan kokeilemiseen liittyvää tietoa.

1.4 Työn rakenne

Työssä tarkastellaan ensin jatkuvaan kokeiluun olennaisesti liittyviä käsitteitä. Näitä ovat toisessa kappaleessa käsiteltävä A/B-testaus ja kolmannessa kappaleessa käsiteltävät jatkuva integraatio sekä jatkuva toimitus. A/B-testaus on jatkuvassa kokeilemisessa keskeisessä asemassa, sillä vaikka muunlaisiakin kokeiluita on olemassa, niin suurin osa kokeiluista on A/B-testejä (Deng ja Shi, 2016). Tästä syystä myös tässä työssä keskitytään pääasiassa A/B-testeillä tehtäviin kokeiluihin. Jatkuva integraatio ja jatkuva

*https://scholar.google.com/
†https://ieeexplore.ieee.org/Xplore/home.jsp
‡https://dl.acm.org/

toimitus tähtäävät siihen, että kehitettävästä ohjelmistosta saadaan jatkuvasti julkaistua käyttäjille lisäarvoa tuottavia uusia toiminnallisuuksia. Jatkuvan kokeilemisen voidaan katsoa olevan seuraava askel näihin jatkuvaan käyttäjille lisäarvon toimittamiseen tähtääviin ohjelmistokehitysmenetelmiin.

Työn neljännessä kappaleessa käsitellään ensin jatkuvaan kokeilemiseen siirtymistä ja millaisia vaatimuksia siirtymisen onnistumiseksi kohdistuu organisaatioon ja kehitettävään ohjelmistoon. Tämän jälkeen tarkastellaan jatkuvan kokeilemisen teknisen toteutuksen kannalta keskeisiä osa-alueita: testi- ja kontrolliversioiden variaatioiden toteutusta, datan keräystä, käsittelyä ja analysointia sekä kokeilun kohdetta. Lopuksi käydään läpi yhteenveto työn löydöksistä ja analysoidaan niitä. Lisäksi tarkastellaan mahdollisia jatkotutkimuksen kohteita.

2 A/B-testaus

Tässä kappaleessa käsitellään A/B-testauksen taustoja sekä selostetaan, miten A/B-testaus käytännön tasolla toimii käyttäen esimerkkitapauksia. Samalla havainnollistetaan hyötyjä, joita A/B-testauksella voidaan saavuttaa.

A/B-testauksen perusidea on testata kahta erilaista versiota samasta ohjelmistosta, valita näistä paremmin menestynyt versio ja ottaa se käyttöön. Tällainen testaus ei ole mikään uusi keksintö, vaan samanlaisista kontrolloiduista tutkimuksista on kirjoitettu kattavasti jo 1910-luvulta lähtien (Crook et al., 2009). Silloin Sir Ronald A. Fisher teki tällaisia tutkimuksia maataloustutkimuksen parissa.

Ohjelmistokehityksessä A/B-testauksen suosio on ollut suuressa kasvussa jo yli vuosikymmenen ajan (Ros ja Runeson, 2018). Yritykset ovat huomanneet, että ohjelmiston kehitystiimi tai muut yrityksen omat sidosryhmät eivät välttämättä tiedä, mitä asiakkaat haluavat. Esimerkiksi Etsyn kehitystiimi halusi toteuttaa verkkokauppaansa muutoksen, jossa käyttäjän klikatessa tuotelistauksesta tuotetta sen sivu avautuu kokonaan uudessa välilehdessä. Tämä idea sai alkunsa siitä, miten kehittäjät itse käyttivät Etsyn verkkokauppaa. Muutos testattiin A/B-testillä. Lopputulos oli, että 70% käyttäjistä lähti sivustolta kokonaan, kun tuote avautui uudessa välilehdessä. Muutosta ei ymmärrettävästi ikinä otettu käyttöön [†]. Microsoftilla, jossa uusien ideoiden A/B-testaus on hyvin yleistä, kolmanneksella testatuista muutoksista on positiivinen vaikutus, kolmanneksella ei ole lainkaan vaikutusta ja kolmanneksella on haitallisia vaikutuksia kehitettävälle ohjelmistolle [‡]. Googlella testatuista muutoksista positiivisia vaikutuksia on vain noin 10%:lla (Kohavi, Deng, Frasca et al., 2013).

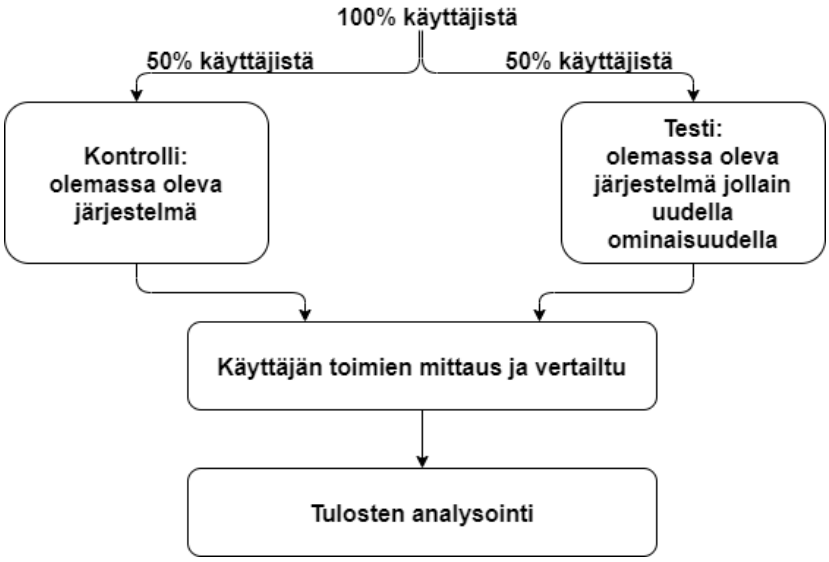
Erityisesti Internetin ja sen myötä selainpohjaisten ohjelmistojen suosion valtava kasvu on tehnyt A/B-testausesta paljon aiempaa helpompaa. Nykypäivänä A/B-testauksesta on varsinkin selainsovelluskehityksessä tullut yleisempää ja sitä käyttävät laajasti monet isot ohjelmistoalan yritykset kuten Microsoft, Amazon, Google, Facebook, eBay ja LinkedIn (Deng ja Shi, 2016). Tästä huolimatta Schermannin ja kumppaneiden

[†]Design for Continuous Experimentation , https://www.slideshare.net/danmckinley/design-for-continuous-experimentation, luettu 13.11.2019

[‡]The Surprising Power of Online Experiments , https://hbr.org/2017/09/the-surprising-power-of-online-experiments, luettu 10.11.2019

teettämän kyselyn mukaan alle puolet yrityksistä tekevät A/B-testausta tai vastaavia kokeiluja käyttäjillään (Schermann, Cito ja Leitner, 2018).

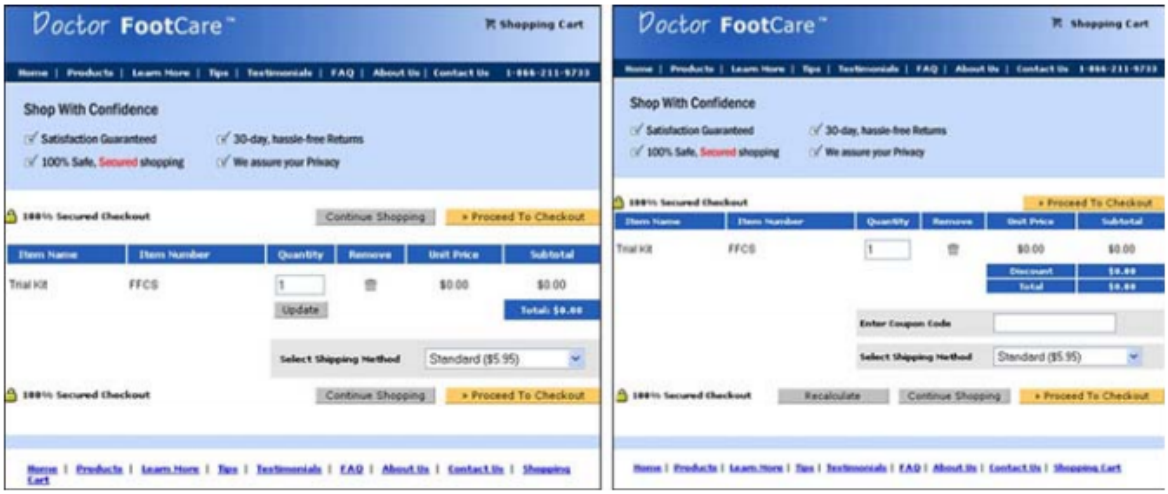
Kuva 2.1 havainnollistaa yksittäisen A/B-testin kulkua. Kontrolliversio on järjestelmän nykyinen käytössä oleva versio ilman mitään muutoksia ja testiversio sisältää yhden tai useamman muutoksen nykyiseen järjestelmään. Käyttäjät jaetaan jollakin menetelmällä satunnaisesti näiden kahden eri version välillä. Kuvassa käyttäjät jaetaan tasaisesti kontrolliversion ja testiversion välillä, mutta käyttäjiä voidaan myös jakaa eri suhteessa. Tässä esimerkissä testiversioita on vain yksi, mutta testiversioita voi myös olla käytössä useampi samaan aikaan. Käyttäjien tekemiä toimintoja mitataan molemmissa versioissa. Kun dataa on kertynyt tarpeeksi, niin testin tulokset analysoidaan. Testiversio otetaan käyttöön, mikäli se suoriutui kontrolliversiota paremmin, jolloin testiversiosta tulee uusi kontrolliversio ja vanha kontrolliversio hylätään.



Kuva 2.1: A/B-testauksen yleiskatsaus (Crook et al., 2009)

Yksittäisen A/B-testin suuruusluokka voi vaihdella paljon. Pienimmillään testi voi kohdistua esimerkiksi käyttöliittymän yksittäisen napin väriin tai jonkin tekstin muutokseen ja suurimmillaan sovelluksen koko käyttöliittymä saattaa olla erilainen testiversion ja kontrolliversion välillä. Palvelinpuolen toiminnallisuuden testaus on myös yleistä. Tällöin testaus voi kohdistua esimerkiksi käyttäjille näytettävän sisällön personointiin tai hakukoneen algoritmin optimointiin ja moniin muihin ominaisuuksiin (Fabijan, Dmitriev, Olsson et al., 2017b). Vaikka yleensä testien kohteina ovat ohjelmiston toiminnallisuudet, niin myös esimerkiksi liiketoimintamalleja sekä erilaisia kampanjoita voidaan testata A/B-testeillä (Fagerholm et al., 2017).

Kuva 2.2 havainnollistaa Doctor FootCare -verkkopalvelun käyttöliittymään tehtyä A/B-testiä. Vasemmanpuoleinen kuva on järjestelmän kontrolliversio ja oikeanpuoleinen testiversio. Tässä tapauksessa testiversio sisältää yhdeksän erilaista muutosta kontrolliversioon nähden. Testin lopputuloksena kuitenkin huomattiin, että testiversio laski palvelun myyntiä jopa 90%. Tämä johtui testiversion muutoksesta, jossa käyttöliittymään lisättiin kenttä kuponkikoodille. Tämä kenttä sai käyttäjät epäröimään ostostaan, jos heillä ei ollut kuponkikoodia. Kuponkikoodikentän poiston jälkeen testiversio paransi myyntiä 6,5% kontrolliversioon nähden. Tämä on hyvä esimerkki siitä, miten todella pieneltä vaikuttavalla muutoksella voi olla suuri vaikutus käyttäjien käyttäytymiseen ja sen myötä yrityksen liiketoimintaan ja miten tärkeää testaus on liiketoiminnan kannalta.



Kuva 2.2: Doctor FootCare -verkkopalvelun A/B-testi (Kohavi, Longbotham et al., 2009)

2.1 Kokonaisarviointikriteerit

Ennen A/B-testin aloittamista testille on oltava jokin tavoite. Tavoitellaanko ohjelmistolle lisää käyttäjiä, lisää ostotapahtumia, parempaa käyttäjien pysyvyyttä vai tehokkaampaa algoritmia suorittamaan jotakin laskentaa? A/B-testauksen tavoitteita kutsutaan yleisesti kokonaisarviointikriteereiksi. Yrityksen A/B-testauksen alkuvaiheessa kokonaisarviointikriteerit saattavat sisältää yksittäisiä edellä mainittuja tavoitteita (Fabijan, Dmitriev, McFarland et al., 2018). Tällaisilla irrallisilla ja yksittäisillä tavoitteilla on kuitenkin vaarana huonontaa ohjelmistoa jollakin toisella osa-alueella. Kielteinen vaikutus ei myöskään välttämättä näy heti, vaan se saattaa näkyä vasta pidemmän

ajan kuluttua. Tämän takia on tärkeää siirtyä pysyvämpiin kokonaisarviointikriteereihin (Kohavi ja Longbotham, 2017).

Parhaassa tapauksessa kokonaisarviointikriteerit sisältävät liiketoiminnan ja testattavan sovelluksen pitkän aikavälin tavoitteita, joissa on otettu huomioon liiketoiminnan tärkeimmät tavoitteet, niihin vaikuttavat tekijät sekä eri tavoitteiden ristiriidat. Ristiriita voi olla esimerkiksi verkkopalvelun mainosten lisääminen, joka lyhyellä aikavälillä lisää tuottoja, mutta huonontuneen käyttäjäkokemuksen takia saattaa pitkällä aikavälillä karkottaa käyttäjiä. Käyttäjät siirtyvät kilpailijoiden palveluihin ja alkuperäinen muutos, joka lisäsi mainoksia, vähentääkin tuottoja merkittävästi (Fabijan, Dmitriev, Olsson et al., 2017a). Kokonaisarviointikriteereihin valittujen tavoitteiden täytyy kuitenkin olla mitattavissa lyhyellä aikavälillä, koska A/B-testit saattavat olla kestoltaan melko lyhyitä. Kohavi ja Longbotham määrittävät tämän aikavälin kahdeksi viikoksi (Kohavi ja Longbotham, 2017).

A/B-testauksella voidaan saavuttaa merkittäviä etuja tuottamalla paremmin asiakkaiden tarpeisiin vastaavia ohjelmistoja. A/B-testaus ei kuitenkaan ota kantaa itse ohjelmistokehitysprosessiin, jolla on hyvin keskeinen rooli siinä, miten nopeasti ja tehokkaasti asiakkaalle saadaan toimitettua uusi ohjelmisto tai uusia toiminnallisuuksia vanhaan ohjelmistoon. Seuraavaksi käsitellään kahta ohjelmistokehitysprosessin käytäntöä, jotka edesauttavat näitä osa-alueita.

3 Jatkuva integraatio ja jatkuva toimitus

Tässä kappaleessa tarkastellaan kahta laajasti käytössä olevaa ohjelmistokehityksen menetelmää, joita käyttämällä pyritään tehostamaan ja parantamaan ohjelmistokehitystä. Nämä menetelmät ovat jatkuva integraatio ja jatkuva toimitus.

3.1 Jatkuva integraatio

Jatkuvan integraation keskeinen ajatus on jatkuva ja tasaisin väliajoin tapahtuva uuden koodin liittäminen nykyiseen koodipohjaan. Tällä vähennetään kehittäjien tekemien muutosten välisten konfliktien määrää sekä konflikteista aiheutuvien bugien määrää (Vasilescu et al., 2015; Elbaum et al., 2014). Jatkuva integraatio mainitaan ensimmäisen kerran Grady Boochin toimesta vuonna 1991 (Hilton et al., 2016) ja sittemmin se liitettiin osaksi 1990-luvun lopulla kehitetyn Extreme Programming -ohjelmistokehitysmenetelmän periaatteita. Laajempaan tietoisuuteen jatkuva integraatio kuitenkin nousi Martin Fowlerin vuonna 2000 julkaiseman kirjoituksen "Continuous Integration"[†] myötä (Hilton et al., 2016), jonka jälkeen sen suosio on jatkanut kasvuaan ja on nykypäivänä monien yritysten käytössä (Elbaum et al., 2014). Huomattavaa on myös, että samana vuonna julkaistiin ketterän ohjelmistokehityksen manifesti, jonka menetelmät ovat olennaisessa osassa jatkuvan integraation käyttöönotossa ja jonka tekijöihin Martin Fowler myös lukeutuu (Vasilescu et al., 2015).

Jatkuvassa integraatiossa aina, kun uutta koodia ollaan integroimassa ohjelmiston koodipohjaan, integroitava koodi käännetään ja projektin yksikkö- ja integraatiotestit ajetaan (Lai ja Leu, 2015). Mikäli käännös onnistuu ja testit suoritetaan onnistuneesti, niin uusi koodi liitetään osaksi koodipohjaa. Tässä vaiheessa koodista voidaan myös mahdollisesti tehdä jonkinlaisia staattisia analyysejä, kuten koodin laadun mittausta tai testien kattavuutta. Jatkuvan integraation periaatteiden mukaisesti kaikki nämä askeleet tapahtuvat täysin automaattisesti, joka luonnollisesti vähentää merkittävästi

[†]Continuous Integration (original version), https://martinfowler.com/articles/originalContinuousIntegration.html, luettu 8.1.2020

turhaa manuaalista työtä. Kehittäjät saavat tällä menetelmällä hyvin nopeasti palautetta tuottamansa koodin toimivuudesta ja mahdolliset bugit havaitaan paljon todennäköisemmin jo aikaisessa vaiheessa (Vasilescu et al., 2015; Lai ja Leu, 2015; Hilton et al., 2016).

Olemassa olevan toiminnallisuuden hajottamisen riskin pienentyminen sekä jatkuvasti kaikesta tekemisestä saatava palaute parantaa kehitystiimin itseluottamusta uuden koodin liittämiseen koodipohjaan sekä uuden toiminnallisuuden julkaisemiseen. Hilton ja kumppanit huomioivat, että jatkuvaa integraatiota käyttävät kehitystiimit tekevät uusia julkaisuja kaksi kertaa enemmän kuin tiimit, joilla ei ole sitä käytössä (Hilton et al., 2016). Vasilescu ja kumppanit tutkivat avoimen lähdekoodin projekteja ja havaitsivat, että koodimuutoksia hyväksyttiin jopa 20,4% enemmän ja hylättiin 42,3% vähemmän, kun projektissa oli käytössä jatkuva integraatio (Vasilescu et al., 2015).

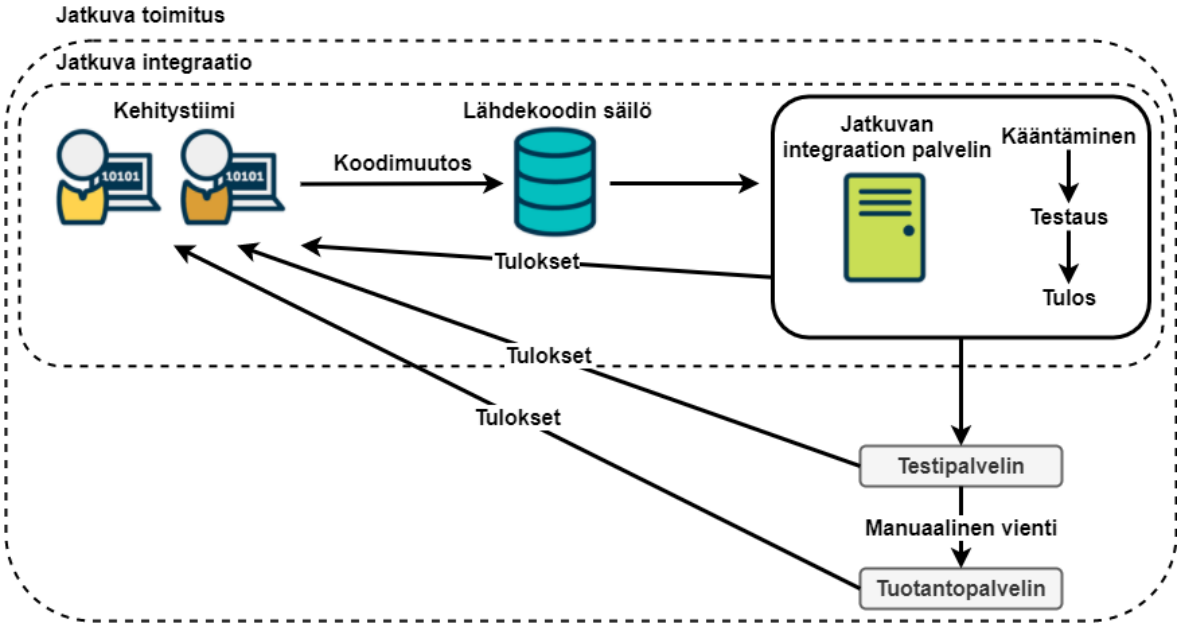
Jatkuvan integraation käyttöönotto on myös vuosien varrella helpottunut merkittävästi. Toteutuksen kannalta olennaiset osat ovat keskitetty versionhallinta, jossa ohjelmiston koodia säilytetään ja hallitaan, sekä jokin jatkuvan integroinnin työkalu. Työkalutuki jatkuvalle integraatiolle on nykypäivänä laajaa, työkaluja on peräti yli 40 erilaista, joista valita (Hilton et al., 2016). Suosittuja työkaluja ovat esimerkiksi Bamboo *, Jenkins † ja Travis CI ‡.

3.2 Jatkuva toimitus

Jatkuva toimitus on luonnollinen jatkumo jatkuvalle integraatiolle ja se esiteltiin ensimmäisen kerran vuonna 2010 kirjassa ”Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”(Farley ja Humble, 2010). Kun jatkuvan integraation periaatteilla on ensin automatisoitu uuden koodin kääntäminen, testaus ja analysointi, niin jatkuvan toimituksen keskeisenä ajatuksena on tämän jälkeen automatisoida uuden koodin vieminen testi- ja tuotantoympäristöihin. Kuva 3.1 havainnollistaa jatkuvan integraation ja jatkuvan toimituksen suhdetta. Kuvasta nähdään miten jatkuva toimitus sisältää jatkuvan integraation kokonaisuuden, mutta siihen kuuluu lisäksi koodimuutosten julkaiseminen testi- sekä tuotantopalvelimille. Manuaalisen tuotantoonviennin vaihe voi monissa tapauksissa myös sisältää jonkinlaista manuaalis-

*Bamboo, https://www.atlassian.com/software/bamboo, luettu 8.1.2020
†Jenkins, https://jenkins.io/, luettu 8.1.2020
‡Travis CI, https://travis-ci.org/, luettu 8.1.2020

ta hyväksymistestausta. Olennaista jatkuvan toimituksen kokonaisuudessa on myös se, miten kehitystiimi saa prosessin jokaisessa vaiheessa ajantasaista palautetta tehtyjen muutosten toimivuudesta ja vaiheen onnistumisesta.



Kuva 3.1: Jatkuva integraatio ja jatkuva toimitus, muokattu*

Tuotantoon julkaisemisen osalta aihetta käsittelevä kirjallisuus ei ole täysin yksimielinen siitä pitääkö jatkuvassa toimituksessa tuotantoonviennin olla myös automatisoitua. Joissain tapauksissa on käytetty termiä jatkuva toimitus sellaiselle järjestelmälle, jossa tuotantoonvienti on manuaalinen ja jatkuva käyttöönotto taas järjestelmälle, jossa tuotantoonvientikin on automatisoitu (Shahin et al., 2017). Suurin osa kirjallisuudesta vaikuttaa kuitenkin käyttävän molemmissa näissä tapauksissa termiä jatkuva toimitus.

Jatkuvan toimituksen on osoitettu antavan merkittäviä etuja sitä käyttäville tiimeille ja organisaatioille. Jatkuva toimitus painottaa pienempiä julkaisuja, joka nopeuttaa ohjelmiston julkaisusykliä. Ohjelmistokehittäjät ovat huomanneet tämän parantaneen koodin laatua ja vähentäneen bugien määrää (Leppänen et al., 2015). Luonnollisesti lisääntynyt automaatio on myös vähentänyt manuaalisen työn määrää, joka on parantanut ohjelmistokehityksen tehokkuutta. Neelyn ja Steelyn (Neely ja Stolt, 2013) esimerkkitapauksessa eräs yritys siirtyi kahdeksan viikon pituisesta julkaisusyklistä jatkuvan toimituksen malliin, jossa uuden julkaisun pystyi tekemään milloin tahansa. Aiem-

*The Product Managers' Guide to Continuous Delivery and DevOps, https://www.mindtheproduct.com/what-the-hell-are-ci-cd-and-devops-a-cheatsheet-for-the-rest-of-us/, luettu 10.1.2020

massa toimintamallissa julkaisusyklin alkuvaiheessa toteutettu toiminnallisuus joutui odottamaan julkaisua pitkään, pahimmillaan lähes kahdeksan viikkoa. Bugikorjausten kohdalla tämä pitkä odotus oli usein erityisen ongelmallinen. Joskus bugikorjauksia vietiin tuotantoon tavallisen julkaisusyklin ohi, mikä aiheutti paljon ylimääräistä manuaalista työtä, joka oli pois uuden toiminnallisuuden toteuttamisesta. Jatkuvan toimituksen mahdollistama milloin tahansa tapahtuva uuden toiminnallisuuden julkaiseminen paransi merkittävästi yrityksen kykyä vastata asiakkaiden tarpeisiin nopeasti ja tehokkaasti.

On kuitenkin huomioitava, että jatkuva toimitus on merkittävästi työläämpi kokonaisuus toteuttaa kuin jatkuva integraatio, jonka takia sitä ei ole otettu käyttöön yhtä laajasti (Leppänen et al., 2015). Jatkuvan toimituksen käyttöönotto asettaa myös omat vaatimuksensa yrityskulttuurille sekä ohjelmistoarkkitehtuurille. Arkkitehtuurin on tuettava automaattista testausta sekä julkaisua, joka tarkoittaa pienemmistä osista koostuvaa modulaarista ohjelmistoarkkitehtuuria (Humble, 2017). Jos ohjelmistoa ei ole alusta alkaen tehty tällaisella arkkitehtuurilla, niin siirtyminen siihen voi olla erittäin työläs ja pitkä prosessi. Esimerkiksi teknologiayhtiö Amazonilla siirtyminen modulaariseen ohjelmistoarkkitehtuuriin vei neljä vuotta (Humble, 2017). Jatkuvan toimituksen toteuttaminen voi myös olla hankalampaa joidenkin alojen järjestelmiin, kuten esimerkiksi terveydenhoitoalan. Kyseisiin järjestelmiin saattaa liittyä paljon erilaista sääntelyä, joka voi vaikeuttaa jatkuvan toimituksen toteutusta merkittävästi (Leppänen et al., 2015).

Jatkuvan integraation ja jatkuvan toimituksen menetelmien käyttöönotolla voidaan siis parantaa merkittävästi kykyä toimittaa asiakkaille lisäarvoa jatkuvasti ja mahdollisimman pienellä vaivalla. Tehokkuus ei kuitenkaan yksinään riitä, vaikka se olisi huippuluokkaa, vaan tuotetun ohjelmiston on myös vastattava asiakkaiden tarpeita. Seuraavaksi käsitelläänkin sitä, miten nämä jatkuvan lisäarvon tuottamisen periaatteet yhdistetään A/B-testauksella saavutettavaan parempaan asiakkaiden tarpeisiin vastaamiseen.

4 Jatkuva kokeileminen

Jatkuva integraatio ja jatkuva toimitus pyrkivät parantamaan ohjelmistokehittäjien työtapoja ja vähentämään riskejä painottamalla jatkuvasti tapahtuvaa toimintaa. Isojen ja hankalien integraatioiden sijaan tehdään jatkuvasti pieniä ja verrattain ongelmattomampia integrointeja. Isojen kokonaisuuksien julkaisemisen sijaan tehdään jatkuvasti pieniä ja vähemmän toiminnallisuutta sisältäviä julkaisuja. Näiden menetelmien hyödyt voivat kuitenkin mennä hukkaan, jos kehitettävä ohjelmisto ei vastaa sen käyttäjien tarpeita. Tämän ongelman jatkuva kokeileminen pyrkii ratkaisemaan tuomalla käyttäjillä tehtävät kokeilut keskeiseksi osaksi ohjelmistokehitysprosessia. Jatkuvan kokeilemisen voidaan katsoa olevan osa luonnollista jatkumoa jatkuvalle integraatiolle ja jatkuvalle toimitukselle. Kaikkien näiden menetelmien perimmäisenä tarkoituksena on tuottaa asiakkaille mahdollisimman laadukasta ohjelmistoa mahdollisimman tehokkaasti, vaikka menetelmät keskittyvätkin eri osa-alueisiin tämän saavuttamiseksi.

Tässä kappaleessa tarkastellaan ensin siirtymistä yksittäisten A/B-testien tekemisestä jatkuvaan kokeilemiseen ja niitä seikkoja, joita on otettava huomioon ennen tämän siirtymisen tekemistä. Tämän jälkeen käsitellään jatkuvan kokeilemisen teknisen toteutuksen kannalta keskeisiä kysymyksiä: miten sovelluksesta eriytetään testi- ja kontrolliversiot? Miten käyttäjät ohjataan näihin eri versioihin? Miten datan keräys eroaa tavallisesti ohjelmistoissa tehtävästä lokituksesta? Miten dataa muunnellaan ja analysoidaan keräyksen jälkeen? Sekä millaisiin kokonaisuuksiin kokeilujen tulisi kohdistua?

4.1 Siirtyminen jatkuvaan kokeilemiseen

Yksittäisillä A/B-testeillä voidaan saada paljon hyödyllistä tietoa tiettyihin ohjelmiston kehittämisen aikana esiin tuleviin kysymyksiin ja kehittää ohjelmistoa vastaamaan paremmin käyttäjien tarpeita. Nämä yksittäiset testit eivät kuitenkaan usein sellaisenaan ole vielä keskeinen osa ohjelmistokehitystä. Testien tai kokeilujen liittäminen keskeiseksi osaksi ohjelmistokehitystä mahdollistaa entistä suuremmat hyödyt ja ohjelmiston kehittämisen vastamaan käyttäjien tarpeisiin entistä paremmin. Tämä jatkuvaan kokeilemiseen siirtyminen vaatii kuitenkin merkittävää panostusta sitä toteuttavalta

organisaatiolta, sillä jatkuvan kokeilemisen toteuttaminen ei ole aivan yksinkertainen toimenpide. Ennen siihen ryhtymistä onkin syytä huolehtia, että tietyt ennakkoehdot täyttyvät sekä liiketoiminta- ja organisaatiotasolla että teknisestä näkökulmasta (Kohavi, Deng, Frasca et al., 2013; Fabijan, Dmitriev, McFarland et al., 2018).

4.1.1 Liiketoiminta- ja organisaatiotason vaatimukset

Luonnollisesti on tärkeää, että organisaatiosta löytyy halua tehdä päätöksiä datan pohjalta sen sijaan, että ne tehtäisiin eri sidosryhmien mielipiteiden pohjalta. Tähän kuuluu olennaisesti ymmärrys siitä, että emme yleisesti ottaen ole kovin hyviä arvioimaan sitä, miten hyviä omat ideamme ovat*. Tästä syystä organisaatiossa on tiedostettava, että jatkuvan kokeilemisen toteutuksen myötä monet toteutetuista uusista ominaisuuksista tai muutoksista paljastuvat kokeiluissa negatiivisiksi tai niillä ei ole mitään vaikutusta eivätkä ne siten päädy ikinä käyttöön asti (Fabijan, Dmitriev, McFarland et al., 2018). Monille tämä voi aluksi tuntua siltä, että tehdään paljon täysin turhaa työtä. Onkin tärkeää, että ajatusmaailma ohjelmistokehitystiimeissä ja organisaatiossa muuttuu pitämään onnistuneesta kokeilusta saatua tietoa arvokkaana, vaikka kokeilun kohteena ollut muutos hylättäisiinkin.

Organisaatiossa on myös tiedostettava, että jatkuva kokeileminen tulee aiheuttamaan lisäkustannuksia (Kohavi, Deng, Frasca et al., 2013). Lisäkustannuksia tulee esimerkiksi jatkuvan kokeilemisen osaamisen hankkimisesta joko kouluttamalla nykyisiä työntekijöitä tai palkkaamalla uusia, joilta osaamista löytyy jo valmiiksi. Esimerkiksi Microsoft tarjoaa työntekijöilleen sisäisiä tilastotieteeseen ja kokeilujen suunnitteluun keskittyviä kursseja (Schermann, Cito ja Leitner, 2018). Useat lähteet huomioivat, että yritys tarvitsee datatieteilijöitä tai vastaavaa osaamista omaavia työntekijöitä (Fabijan, Dmitriev, Olsson et al., 2017b; Fagerholm et al., 2017). Erityisesti osaamista tarvitaan oletamuksien testauksen, satunnaistamisen, otantakokojen ja luottamusvälien saralla. Myös uudet tekniset hankinnat todennäköisesti tulevat aiheuttamaan kustannuksia. Tällaisia voivat olla esimerkiksi lisäpalvelimet tai uudet pilvipalvelut kokeilujen ajamiseen.

Kokonaisarviointikriteerit ovat keskeisessä roolissa jatkuvassa kokeilemisessa. Kokeilemisen alkuvaiheessa on tärkeää testata erilaisia kokonaisarviointikriteereitä, jotta saadaan tärkeää osaamista niiden muodostamisesta (Fabijan, Dmitriev, McFarland

*The Surprising Power of Online Experiments, https://hbr.org/2017/09/the-surprising-power-of-online-experiments, luettu 13.1.2020

et al., 2018). Pidemmälle viedyssä jatkuvassa kokeilemisessa on kuitenkin syytä olla huolella laaditut kokonaisarviointikriteerit kehitettävien ohjelmistojen osalta (Kohavi, Deng, Frasca et al., 2013). Kokonaisarviointikriteerit on siis syytä tehdä tuotteen pitkäaikaisten tavoitteiden mukaisesti, mutta kuitenkin siten, että muutokset niihin ovat mitattavissa lyhyellä, jopa kahden viikon aikavälillä.

Ohjelmistokehityksen pitäisi tapahtua iteratiivisesti ja alun perin Agile Manifestossa* esiteltyjä periaatteita noudattaen. Jatkuva kokeileminen ei sovellu vesiputousmallilla tehtävään ohjelmistokehitykseen†, sillä vesiputousmallissa ohjelmistokehitys ei tapahdu iteraatioissa, vaan tarkasti ennalta määritellyissä vaiheissa. Tällöin kokeilu voisi tapahtua vasta, kun ohjelmiston kehitys on valmis ja ohjelmisto on jo käytössä. Vesiputousmallilla tehtävässä ohjelmistokehityksessä tässä vaiheessa ei kuitenkaan ole enää tarkoitus tehdä siihen muutoksia.

Jatkuvan kokeilemisen käyttöönottoa varten kokeiluja tekevillä työntekijöillä on oltava pääsy tuotteesta kerättyyn dataan. Joissain tapauksissa, kuten hyvin arkaluonteisen tai tietoturvaluokitellun datan tapauksessa, tämä voi olla hankalaa tai jopa mahdotonta (Fabijan, Dmitriev, Olsson et al., 2017b). Myös turvallisuuskriittisissä järjestelmissä jatkuvan kokeilemisen toteuttaminen voi olla hyvin hankalaa. Tähän lopputulokseen päätyivät Giaimo ja kumppanit (Giaimo ja Berger, 2017) tutkiessaan jatkuvan kokeilemisen toteuttamista itseajavien autojen ohjelmistoon.

Kun ennakkoehdot kokeiluiden tekemiselle täyttyvät, niin on aika ottaa ensiaskel jatkuvan kokeilemisen toteuttamisessa. Tässä vaiheessa erillistä kokeiluiden suunnitteluun ja toteuttamiseen keskittyvää tiimiä ei vielä tarvita, vaan muutama asiantuntija riittää (Fabijan, Dmitriev, McFarland et al., 2018). Nämä voidaan joko palkata kokonaan uusina työntekijöinä tai kouluttaa nykyisistä työntekijöistä. Kokeiluiden suunnitteluun ja toteuttamiseen erikoistuneet asiantuntijat tekevät kehitystiimeille kokeiluita ja samalla jakavat osaamistaan kehitystiimeille. Alkuvaiheen kokeiluiden olisi suositeltavaa kohdistua yksittäisten toiminnallisuuksien kokeilemiseen.

Taulukko 4.1 tiivistää jatkuvan kokeilemisen alkuvaiheen toteutuksen organisaatiotason osa-alueita ja minkä tasoista toteutusta kullekin osa-alueelle olisi syytä tavoitella. Fabijan ja kumppanit kutsuvat tätä siirtymistä kokeiluiden tekemiseen ryömimisvaiheeksi ja kuvaavat ryömimisvaiheen lisäksi kävelemis-, juoksemis- ja lentämisvaiheita, mutta

*Manifesto for Agile Software Development, https://agilemanifesto.org/, luettu 9.1.2020

†Design for Continuous Experimentation, https://www.slideshare.net/danmckinley/design-for-continuous-experimentation, luettu 20.11.2019

niitä ei sisällytetty tähän, sillä tässä keskitytään vain kokeiluiden tekemiseen siirtymiseen (Fabijan, Dmitriev, McFarland et al., 2018).

Osa-alue	Toteutuksen taso
Kehitystiimien omavaraisuus	**Rajallista:** Kehitystiimit ovat riippuvaisia kokeiluiden tekemiseen erikoistuneista asiantuntijoista
Kokeiluista vastaavan tiimin organisoiminen	**Ei organisoitumista:** ei erillistä tiimiä kokeiluiden tekemiseen
Kokonaisarviointikriteerit	**Kokeilukohtaiset:** Kokonaisarviointikriteerit määritetään erikseen jokaiselle kokeilulle
Kokeiluiden vaikutukset	**Yksittäiset toiminnallisuudet:** Kokeileminen kohdistuu yksittäisien uusien ominaisuuksien hyödyn testaamiseen

Taulukko 4.1: Jatkuvan kokeilemisen alkuvaiheen organisaatiotason osa-alueet, muokattu (Fabijan, Dmitriev, McFarland et al., 2018)

4.1.2 Tekniset vaatimukset

Organisaatioon ja työntekijöiden ajatusmaailmaan kohdistuvien ennakkoehtojen lisäksi jatkuvan kokeilemisen käyttöönotto asettaa yrityksille myös teknisiä vaatimuksia. Erityistä kokeilujen tekemiseen tarkoitettua sovellusalustaa ei ole tarpeen olla jatkuvaan kokeiluun siirryttäessä (Fabijan, Dmitriev, McFarland et al., 2018), mutta yrityksellä on kuitenkin oltava valmiudet tehdä yksittäisiä A/B-testejä. Näiden testien tulosten luotettavuuden tulee myös olla varmistettu (Kohavi, Deng, Frasca et al., 2013).

Jatkuva integraatio ja jatkuva toimitus ovat olennaisessa osassa jatkuvan kokeilemisen käyttöönotossa. Jatkuva integraatio ja jatkuva toimitus mahdollistavat ohjelmistoon tehtävien kokeilujen ketterän ja joustavan kehittämisen sekä edesauttavat kokeiluissa käytettävien eri variaatioiden julkaisemista mahdollisimman pienellä vaivalla. Jatkuvan kokeilemisen voidaankin katsoa olevan jatkoa ohjelmistoalan pidempiaikaiselle suunnalle, jossa pyritään julkaisemaan käyttäjille tasaisesti ja jatkuvasti muutoksia ja uusia toiminnallisuuksia.

Jatkuva integraatio ja jatkuva toimitus edellyttävät ohjelmiston arkkitehtuurilta modulaarisuutta ja koska jatkuva kokeileminen on osaksi riippuvainen näistä, niin luonnollisesti myös jatkuvan kokeilemisen toteutus on riippuvainen ohjelmiston arkkitehtuurista tässä suhteessa. Lisäksi monoliittistä arkkitehtuuria noudattaviin ohjelmistoihin saattaa usein olla selvästi hankalampaa kohdistaa kokeiluita kuin modulaarisen

mikroservice-arkkitehtuurin mukaan rakennettuun ohjelmistoon (Schermann, Cito ja Leitner, 2018). Tämä on ainakin vielä monien yritysten kompastuskivi jatkuvan kokeilemisen toteutuksessa. Schermann ja kumppanit (Schermann, Cito ja Leitner, 2018) teettivät kyselyn, jossa kysyttiin miksi yritys ei ole ottanut käyttöön jatkuvaa kokeilemista. Vastaajien yleisin syy oli kokeiltavan ohjelmiston arkkitehtuurin soveltumattomuus kokeiluihin.

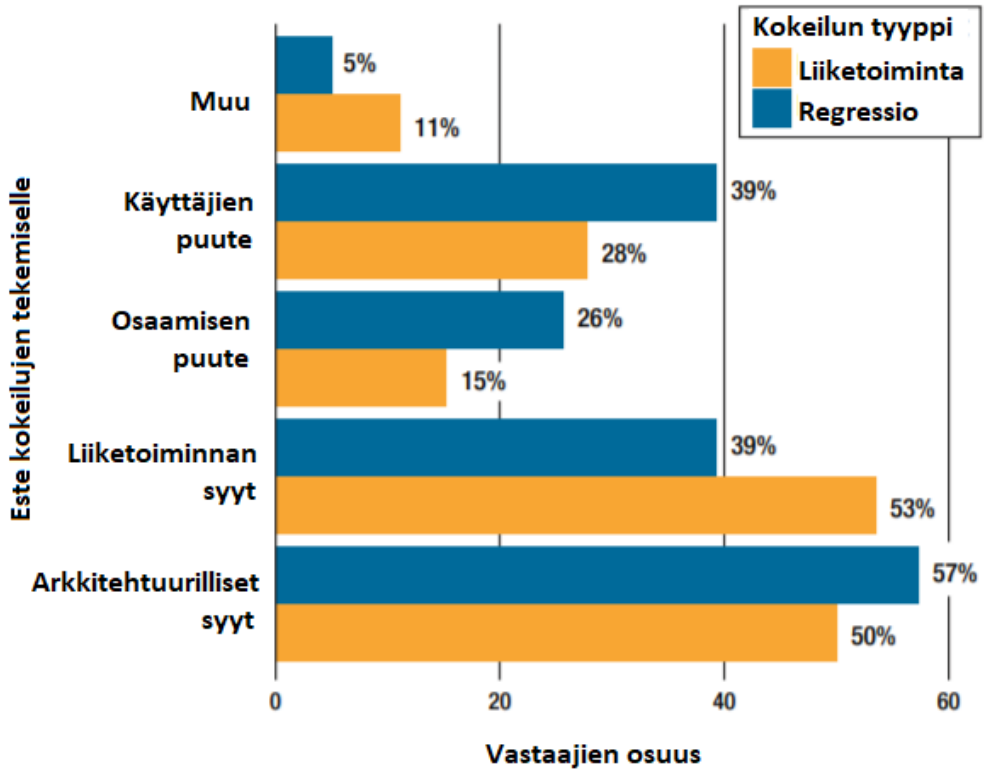
Jokaisessa kokeilussa on oltava mukana tietty vähimmäismäärä käyttäjiä, jotta sen tulosten luotettavuus on riittävällä tasolla. Tämän takia liian vähäinen käyttäjämäärä voi myös estää kokeiluiden tekemistä (Schermann, Cito, Leitner et al., 2019). Tätä ongelmaa esiintyy erityisesti startup-yritysten keskuudessa, sillä startup-yrityksillä ei usein ole isoa määrää vakiintuneita käyttäjiä.

Myös kokeilun kohteena olevan ohjelmiston asiakkaat voivat vaikuttaa jatkuvan kokeilemisen käyttöönottoon. Rissanen ja kumppanit (Rissanen ja Münch, 2015) tutkivat jatkuvan kokeilemisen käyttöönottoa yritysasiakkaille suunnattuja ohjelmistoja kehittävässä yrityksessä. Yrityksen ohjelmistoja käytti vain pieni määrä käyttäjiä, joka vaikeutti luotettavien kokeiluiden tekemistä. Lisäksi osaa yrityksen ohjelmistoista käytettiin vain rajapintojen kautta sen sijaan, että niitä olisi käytetty jonkinlaisen käyttöliittymän kautta. Tällaisten koneellisesti käytettävien rajapintojen testaukseen A/B-testit eivät sovellu, vaan niiden testaukseen on käytettävä esimerkiksi käyttöasteiden mittausta tai alfatestausta. Näistä hidasteista huolimatta tutkimuksen kohteena ollut yritys hyötyi jatkuvan kokeilemisen käyttöönotosta, mutta tätä tapausta ei voida kuitenkaan yleistää muihin yritysasiakkaita palveleviin ohjelmistoihin (Rissanen ja Münch, 2015).

Kuva 4.1 antaa käsitystä siitä, mitä asioita ohjelmistokehittäjät kokevat kokeiluiden tekemisen suurimmiksi esteiksi. Tässä kyselyssä on eroteltu liiketoiminnan kokeilut regressiokokeiluista. Liiketoiminnan kokeiluilla tarkoitetaan tässä uusien ideoiden ja toiminnallisuuksien hyödyllisyyden kokeiluja. Regressioon keskittyvillä kokeiluilla varmistetaan vanhan toiminnallisuuden toimivuus muutosten jälkeen. Arkkitehtuurilliset syyt nousevat kyselyssä suurimmaksi esteeksi, mutta liiketoiminnan syyt yltävät lähes yhtä korkealle. Myös käyttäjien sekä osaamisen puute ovat merkittäviä esteitä isolla osalla ohjelmistokehittäjiä, varsinkin regressioon keskittyvien kokeiluiden tekemiseen.

Taulukko 4.2 tiivistää jatkuvan kokeilemisen teknisiä osa-alueita ja minkä tasoista toteutusta niillä tulisi alkuvaiheessa tavoitella.

Kun tässä kappaleessa esiin nostetut tekniset sekä organisaatiotason vaatimukset on



Kuva 4.1: Jatkuvan kokeilemisen esteet (Schermann, Cito ja Leitner, 2018)

Osa-alue	Toteutuksen taso
Tekninen painopiste	**Datan keräys:** Siirtyminen ongelmien selvittämistä varten tehdystä lokituksesta kontekstuaaliseen lokitukseen
Kokeilemisalustan vaatimukset	**Ei vaatimuksia:** Tässä vaiheessa ei ole vielä tarvetta kokeilemisalustalle.
Kokeilemisen kokonaisvaltaisuus	**Näkyvyys:** Muutamia kokeiluita osoittamaan kokeilemisen hyötyjä

Taulukko 4.2: Jatkuvan kokeilemisen alkuvaiheen tekniset osa-alueet, muokattu (Fabijan, Dmitriev, McFarland et al., 2018)

otettu huomioon, niin seuraavaksi kysymykseksi nousee itse jatkuvan kokeilemisen toteuttaminen. Seuraava kappale paneutuu erilaisiin menetelmiin, joilla sovelluksesta voidaan toteuttaa erilaisia variaatioita ja siihen, miten käyttäjät jaetaan näihin eri variaatioihin.

4.2 Testi- ja kontrolliversioiden variaatioiden toteutus

Jokaisessa kokeilussa käyttäjät pitää jakaa kontrolliversion ja yhden tai useamman testiversion välillä. Tätä varten tarvitaan ensinnäkin jokin menetelmä, jolla saadaan toteutettua sovelluksesta halutut erilaiset variaatiot, eli kontrolli- ja testiversiot. Lisäksi tarvitaan menetelmä, jolla käyttäjät jaetaan käyttämään näitä eri variaatioita. Tämän toteutuksessa on myös tärkeää huomioida se, miten pidetään kirjaa siitä, mihin ryhmään mikäkin käyttäjä kuuluu.

4.2.1 Ominaisuuskytkimet

Ominaisuuskytkin (eng. feature toggle) on menetelmä, jossa jokin sovelluksen ominaisuus on päällä tai pois päältä riippuen sovelluksen konfiguraatiossa asetetun muuttujan arvosta. Ominaisuuskytkimiä on käytetty jo pidemmän aikaa jatkuvan integraation toteutuksen apuna, sillä niiden avulla keskeneräisenkin toiminnallisuuden voi integroida ohjelmistoon (Neely ja Stolt, 2013). Tällöin keskeneräinen toiminnallisuus yksinkertaisesti kytketään pois päältä, kunnes se on valmis. Tämä ei näy käyttäjille mitenkään.

Kuvasta 4.2 nähdään ominaisuuskytkimien toimintaperiaate yksinkertaisimmillaan. Tässä esimerkissä yksinkertaisella if-lauseella tarkistetaan kuuluuko kokeilun kohteena olevan toiminnallisuuden olla käytössä tälle käyttäjälle ja suoritetaan sen perusteella joko uuden toiminnallisuuden sisältävä koodi tai vanha, ei muutoksia sisältävä koodi. Ominaisuuskytkimillä voidaan myös toteuttaa tätä esimerkkiä monimutkaisempien kokeilujen logiikka. Esimerkiksi pelkän jonkin ominaisuuden näyttämisen tai piilottamisen sijaan voidaan parametrisoida, miten uusi toiminnallisuus näytetään tai uusi toiminnallisuus näytetään käyttäjälle vain, jos käyttäjä tekee ensin tietyt toimenpiteet (Tang et al., 2010).

Kuvan 4.2 esimerkki ei kuitenkaan ota tarkemmin kantaa siihen, miten ominaisuuskytkimiä käyttävän sovelluksen pitäisi päättää suoritetaanko koodin if- vai else-haara. Tämän logiikan toteuttamiseen on useita erilaisia vaihtoehtoja. Yksinkertainen tapa on käyttää erillistä palvelua, joka sisältää logiikan käyttäjien jakamiseen testi- ja kontrolliryhmiin ja joka tallentaa jokaiselle käyttäjälle määrätyn ryhmän (Gupta et al., 2018). Kutsumalla palvelua ohjelmisto saa vastauksena tiedon siitä, mikä osa koodista suoritetaan. Erillistä palvelua voidaan käyttää yhtä hyvin niin asiakasohjelmistoissa kuin

```
1 if isEnabled('fastCheckout', $user)
2 # uuden toiminnallisuuden sisältävä koodi
3 else
4 # vanhan toiminnallisuuden sisältävä koodi
5 end
```

Kuva 4.2: Ominaisuuskytkin yksinkertaisimmillaan (Schermann, Cito ja Leitner, 2018)

palvelinohjelmistoissa, mutta sen huonona puolena on palvelukutsun aiheuttama viive. Tätä haittaa voidaan kuitenkin minimoida tekemällä ensimmäinen palvelukutsu esimerkiksi sovelluksen käynnistyessä ja tekemällä niitä tämän jälkeen tasaisin väliajoin. Sovellukselle määritettyä ryhmää ei kuitenkaan yleensä vaihdeta kesken suorituksen vaan vasta seuraavalla käynnistyskerralla. Tällä vältetään käyttäjäkokemuksen huonontaminen kesken suorituksen tapahtuvilla muutoksilla.

Pyynnön annotaatio on toinen menetelmä määrittää ominaisuuskytkimien suorittama koodi. Pyynnön annotaatiota käytettäessä ominaisuuskytkimiä käyttävän sovelluksen tekemät palvelukutsut ohjataan jonkin välityspalvelimen läpi, joka määrittää pyynnöt kuulumaan testi- tai kontrolliryhmään ja tallentaa tiedon tästä HTTP-pyynnön otsikoihin (Gupta et al., 2018). Tämän menetelmän huono puoli on sen rajallinen soveltuvuus, sillä se soveltuu lähinnä selainsovellusten käyttäjien ryhmien määrittämiseen.

Ominaisuuskytkimien suorittama koodi voidaan myös määrittää paikallisen kirjaston avulla. Tällöin vältytään erillisen palvelukutsun tekemiseltä, sillä paikallinen kirjasto määrittää suoritettavan koodin konfiguraationsa avulla (Gupta et al., 2018). Konfiguraatio päivitetään keskitetyltä palvelimelta tasaisin väliajoin. Tämä menetelmä on kuitenkin näistä menetelmistä monimutkaisin toteuttaa. Taulukko 4.3 tiivistää vielä tässä käsitellyt erilaiset menetelmät käyttäjien jakamisesta testi- ja kontrolliryhmiin.

Ominaisuuskytkimien toteutuksen avuksi on myös saatavilla kirjastoja sekä palveluita, kuten esimerkiksi LaunchDarkly*, joka muun muassa mahdollistaa sovelluksen ominaisuuskytkimien päälle ja pois laittamisen niille tarkoitetusta verkkopalvelusta. Koska ominaisuuskytkimet ovat osa sovelluksen koodia, niin ne soveltuvat käytettäviksi kaikenlaisiin asiakas- sekä palvelinohjelmistojen kokeiluihin. Joustavuuden lisäksi ominaisuuskytkimet ovat helppoja käyttää ja ne ovatkin erilaisista käytetyistä kokeilumenetelmistä tehdyn kyselyn perusteella suosituin tapa tehdä sovelluksesta eri variaatioita

*LaunchDarkly, https://launchdarkly.com/product/, luettu 17.1.2020

Menetelmä	Soveltuvuus	Hyvät puolet	Huonot puolet
Erillinen palvelu	Asiakasohjelmistot ja palvelinohjelmistot	Yksinkertainen toteuttaa	Palvelukutsu aiheuttaa viiveen
Pyynnön annotaatio	Selainsovellukset	Ei viivettä aiheuttavaa palvelukutsua	Soveltuu lähinnä selainohjelmistoihin
Paikallinen kirjasto	Asiakasohjelmistot ja palvelinohjelmistot	Ei viivettä aiheuttavaa palvelukutsua	Monimutkaisempi toteuttaa kuin muut menetelmät

Taulukko 4.3: Eri menetelmät kokeilujen käyttäjien ryhmien määrittämiseksi (Gupta et al., 2018)

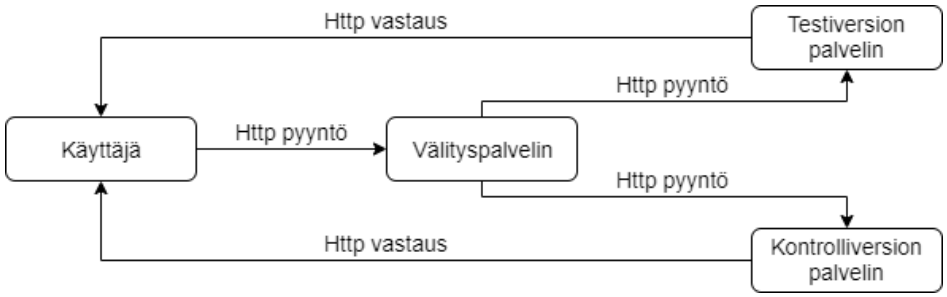
(Schermann, Cito, Leitner et al., 2019).

Sovelluksessa voi olla samanaikaisesti käytössä yksi tai useampi ominaisuuskytkin, mutta on huomioitava, että jokainen lisätty ominaisuuskytkin voi mahdollisesti kaksinkertaistaa erilaisten sovelluksessa syntyvien skenaarioiden määrän. Tästä seuraa se, että iso määrä ominaisuuskytkimiä vaikeuttaa sovelluksen testausta sekä voi helposti aiheuttaa vaikeasti selvitettäviä bugeja. Mitään tiettyä ylärajaa ominaisuuskytkimien määrälle ei ole. Suuntaa-antavana esimerkkinä eräässä yrityksessä oli yhtä aikaisesti käytössä 150 eri ominaisuuskytkintä. Tästä aiheutui merkittäviä ongelmia testauksen ja ilmenneiden bugien saralla, joten määrää vähennettiin radikaalisti (Schermann, Cito, Leitner et al., 2019). Yleisenä ohjenuorana voi pitää sitä, että jos sovelluksen ominaisuuskytkimien luomiseen, ylläpitoon ja poistoon menee merkittävästi aikaa, niin ominaisuuskytkimiä on liikaa*. On myös hyvä muistaa, että jokainen lisätty ominaisuuskytkin lisää ohjelmistoon teknistä velkaa, sillä kokeilun päättymisen jälkeen siihen liittyvän ominaisuuskytkimen koodi täytyy poistaa (Schermann, Cito ja Leitner, 2018).

4.2.2 Ajonaikainen liikenteen jakaminen

Toinen suosittu tapa tarjota käyttäjille eri variaatioita ohjelmistosta on käyttäjien liikenteen jakaminen (eng. traffic splitting) ajonaikaisesti eri variaatioihin ohjelmistosta (Schermann, Cito ja Leitner, 2018). Tällöin testiversiot ovat kokonaan kontrolliversiosta ja toisistaan erillään ajettavia instansseja, joita voidaan suorittaa kokonaan erillisellä palvelimella, erillisellä virtuaalipalvelimella tai saman palvelimen eri portissa (Kohavi, Longbotham et al., 2009). Käyttäjän lähettämät pyynnöt ohjataan ensin välityspalvelimelle, joka sisältää logiikan käyttäjän määrittämiseksi testi- tai kontrolliryhmään sekä tämän tiedon ylläpitämiseen. Ryhmän määrittämisen jälkeen pyyntö

*FeatureToggle, https://martinfowler.com/bliki/FeatureToggle.html, luettu 17.1.2020



Kuva 4.3: Liikenteen ajonaikainen jakaminen (Kohavi, Longbotham et al., 2009)

ohjataan eteenpäin kontrolli- tai testipalvelimelle, joka palauttaa käyttäjälle vastauksen. Kuva 4.3 havainnollistaa käyttäjäliikenteen ohjaamista.

Ajonaikaisessa liikenteen jakamisessa välitys- tai kuormantasauspalvelin nousee kriittiseksi osaksi järjestelmää. Tämän osan toteuttamiseen on muutamia eri lähestymistapoja (Schermann, Cito ja Leitner, 2018). Useat pilvipalveluiden tuottajat tarjoavat tätä valmiina pilvipalveluna. Palvelun käyttäjän tehtäväksi jää ainoastaan konfiguroida palvelu omiin tarpeisiinsa sopivaksi. Tämä lähestymistapa on yksinkertainen ja vaivaton jos järjestelmän muutkin osat ovat osa samaa pilvipalvelua. Liikenteen jakamista valmiina pilvipalveluna tarjoajia ovat esimerkiksi Amazon Web Services*, Google Cloud†, Microsoft Azure‡ ja Cloudflare§.

Hieman työläämpi lähestymistapa liikenteen jakamisen toteuttamiseen on käyttää jotakin siihen tarkoitettua valmista ohjelmistoa. Työläämmän tästä vaihtoehdosta tekee se, että tällä menetelmällä täytyy itse huolehtia ohjelmiston asennuksesta ja ylläpidosta. Tällaisia valmiita ohjelmistoja on sekä ilmaisia avoimen lähdekoodin vaihtoehtoja kuten HAProxy¶ ja Seesaw‖ että lukuisia maksullisia vaihtoehtoja, jotka usein sisältävät paljon muutakin toiminnallisuutta kuin liikenteen jakamisen.

Kontrolli- ja testiversioiden ollessa täysin toisistaan erillisiä ohjelmiston koodiin ei synny ylimääräistä kompleksisuutta kuten ominaisuuskytkimien tapauksessa. Tämän myötä myöhemmin ylimääräistä työtä tai jopa ongelmia aiheuttavaa teknistä velkaa ei myöskään pääse kertymään. Haittapuolena on koko sovelluksen kopioiminen ja molempien versioiden ylläpitäminen samanaikaisesti. Tämä aiheuttaa merkittävästi

*Amazon Web Services, https://aws.amazon.com/, luettu 16.1.2020
†Google Cloud, https://cloud.google.com/, luettu 16.1.2020
‡Microsoft Azure, https://azure.microsoft.com/en-us/, luettu 16.1.2020
§Cloudflare, https://www.cloudflare.com/, luettu 16.1.2020
¶HAProxy, https://www.haproxy.org/, luettu 16.1.2020
‖Seesaw, https://github.com/google/seesaw, luettu 16.1.2020

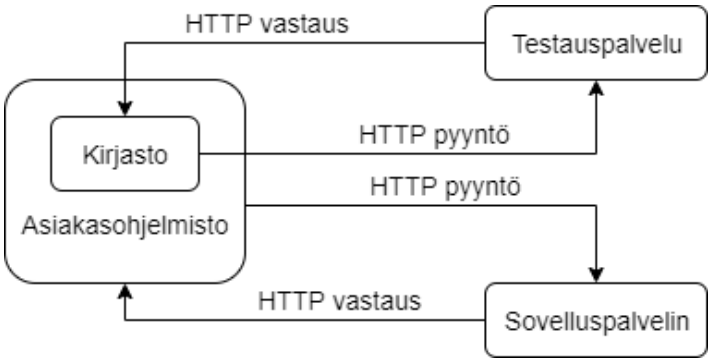
ylimääräisiä kustannuksia tarvittavan palvelinkapasiteetin nousun sekä verkkoliikenteen kasvun myötä (Schermann, Cito ja Leitner, 2018). Tämä muodostuu erityisesti ongelmaksi mikäli halutaan samanaikaisesti ajaa useita testiversioita, sillä jokainen näistä tarvitsee oman instanssinsa. Lisäksi välityspalvelin kasvattaa omalta osaltaan järjestelmän kompleksisuutta. Uusien instanssien luomisen vaivan ja kustannusten takia tätä menetelmää suositellaan käytettäväksi tapauksiin, joissa kokeilun kohteena olevat muutokset ovat kooltaan erityisen isoja. Tällaisia ovat esimerkiksi vanhan järjestelmän korvaaminen kokonaan uudella tai vanhan järjestelmän siirtäminen uudelle alustalle (Kohavi, Longbotham et al., 2009).

4.2.3 Ulkopuolinen palvelu

Sovelluksen variaatioiden toteuttamiseen voidaan myös käyttää ulkopuolista palvelua. Tällöin sovellukseen sisällytetään joko suoraan pätkä koodia tai riippuvuuden kautta jokin kirjasto. Sovellukseen liitetty koodi huolehtii ulkopuolisen palvelun palvelimien kanssa kommunikoimisesta. Ulkopuolinen palvelu palauttaa vastauksen, joka kertoo asiakasohjelmistossa sijaitsevalle koodille ohjeet muutoksista, joita sen kuuluu asiakasohjelmistoon tehdä (Kohavi, Longbotham et al., 2009). Kirjasto huolehtii lisäksi käyttäjän tekemien toimintojen datan keräämisestä sekä tämän datan lähettämisestä palvelimille tallennettavaksi ja käsiteltäväksi.

Ulkopuolisen palvelun käyttö on tarkoitettu lähinnä selainpohjaisiin sovelluksiin. Niiden keskuudessa tämä menetelmä onkin erityisen suosittu helppoutensa ansiosta, sillä monet yritykset tarjoavat tätä menetelmää valmiina palveluna. Näissä yleensä riittää, että sivuston DOM:iin (eng. Document Object Model) lisätään vain valmis pätkä koodia. Tämä koodi lähettää palvelun tarjoajan palvelimelle pyynnön ennen kuin käyttäjän selain mallintaa sivuston käyttäjälle. Palvelin sisältää kaiken testaukseen liittyvän logiikan. Selainohjelmiston toiminnallisuutta muokataan palvelimelta saadun vastauksen perusteella tekemällä muutoksia suoraan ohjelmiston DOM:iin. Kuvasta 4.4 nähdään ulkopuolisen testauspalvelun toimintaperiaate.

Ulkopuolisen testauspalvelun käytön huono puoli on, että muutokset ohjelmistoon pitää tehdä sovelluspalvelimen kutsumisen jälkeen, mutta kuitenkin ennen kuin mitään sivuston osaa mallinnetaan käyttäjän näkyviin (Kohavi, Longbotham et al., 2009). Sivuston mallintaminen siis viivästyy tämän palvelukutsun verran. Tämä voi olla erityisen merkittävää, jos käyttäjä on huonon yhteyden päässä tai jos palvelin on ruuhkautu-



Kuva 4.4: Ulkopuolisen testauspalvelun käyttö

nut. Lisäksi ulkopuolisen testauspalvelun tekemillä DOM:iin kohdistuvilla muutoksilla on aina riskinä hajottaa tai sotkea sivuston normaalia toimintaa, sillä tämä koodi on täysin erillistä sivuston koodista eivätkä nämä tiedä toistensa toiminnasta mitään.

Käsitellyt arkkitehtuuriratkaisut sovelluksen variaatioiden eriyttämismenetelmistä, niiden soveltuvuudesta eri ohjelmistotyyppien kokeilujen toteutukseen ja niiden hyvistä sekä huonoista puolista ovat tiivistettyinä taulukkossa 4.4.

Menetelmä	Soveltuvuus	Hyvät puolet	Huonot puolet
Ominaisuuskytkimet	Kaikki sovellukset	Helppo toteuttaa	Tekninen velka, kompleksisuus jos laajassa käytössä
Liikenteen jakaminen	Selainsovellukset, palvelinsovellukset	Ei koodimuutoksia	Skaalautuvuus, kustannukset
Erillinen palvelu	Selainsovellukset	Helppokäyttöisyys	Viive sivuston mallintamiseen, normaalin toiminnallisuuden hajoamisen riski

Taulukko 4.4: Eriyttämismenetelmät

4.2.4 Käyttäjien jakaminen testi- ja kontrolliryhmiin

Mikäli käytössä ei ole ulkopuolista palvelua hyödyntävää toteutustapaa käyttäjien jakamiseksi testi- ja kontrolliryhmiin, niin tämän toteutus jää kokeiluiden tekijöiden toteutettavaksi. On tarpeellista tarkentaa hieman, mitä tämän toteutus käytännössä pitää sisällään. Hyvin yksinkertainen tapa jakaa käyttäjät kontrolli- ja yhden tai useamman testiversion välillä on satunnaisarvonta (Tang et al., 2010). Tällöin järjestelmä luo jokaisen palvelimelle saapuneen pyynnön kohdalla satunnaisen luvun ja ohjaa sen pe-

rusteella käyttäjän lukua vastaavaan variaatioon. Joissain tapauksissa tämä voi olla toimiva lähestymistapa, mutta suurimmassa osassa kokeiluja käyttäjä halutaan tunnistaa jollain lailla, jotta käyttäjä pysyy pyynnöstä toiseen samassa testi- tai kontrolliversiossa, johon hänet ensimmäisellä pyynnöllä arvottiin. Mikäli pyynnön tekee esimerkiksi jokin sulautettu järjestelmä, järjestelmä johon käyttäjä on kirjautunut sisään tai jokin muu asiakasohjelmisto, joka voidaan varmuudella yksilöidä, niin tunnistaminen on helppoa.

Kokeiluiden kohteena on kuitenkin ylivoimaisesti eniten selainsovellukset ja erityisesti niiden käyttöliittymien ulkoasut (Tang et al., 2010). Tällaisissa kokeiluissa on hyvin yleistä, että käyttäjää ei pystytä tunnistamaan edellä mainituilla keinoilla. Mikäli käyttäjää ei tunnistettaisi millään lailla, niin käyttäjän käyttökokemus saattaisi olla hyvin hämmentävä; hän saattaisi päätyä eri variaatioon ohjelmistosta jokaisella sivulatauksella ja saisi aina erilaisella ulkoasulla varustetun ohjelmiston. Tästä syystä käyttäjät on voitava tunnistaa jollain tavalla, jotta käyttäjälle voidaan tarjota samaa variaatiota sivulatauksesta toiseen.

Tyypillisesti näissä tapauksissa käytetään evästeitä (Tang et al., 2010). Evästeet ovat selainsovellusten käyttäjien laitteisiin tallentamia pieniä määriä dataa sisältäviä tietueita. Evästeitä tarvitaan muun muassa kirjautuneiden käyttäjien tunnistamiseen, erilaisten sivustokohtaisten asetusten tallentamiseen, verkkokauppojen ostoskorin tallentamiseen sekä personalisoidun sisällön tarjoamiseen, kuten esimerkiksi verkkokaupan ostosuositusten näyttämiseen katsottujen tuotteiden perusteella*. Evästeitä on kahta eri tyyppiä: istuntokohtaisia evästeitä ja pysyviä evästeitä. Istuntokohtaiset evästeet poistetaan käyttäjän selaimesta, kun selain suljetaan. Pysyville evästeille asetetaan niiden luontivaiheessa jokin erääntymisaika, jolloin ne poistetaan. Käyttäjä voi myös tyhjentää selaimensa evästeet milloin tahansa. Nykypäivänä evästeet ovat hyvin yleisiä ja suurin osa kaupallisista sivustoista tallentaa käyttäjän selaimeen useita evästeitä.

Monissa jatkuvan kokeilemisen selainsovelluksiin kohdistuvissa kokeiluissa käyttäjän selaimeen tallennetaan eväste, joka sisältää tiedon siitä mihin ohjelmiston variaatioon käyttäjä on arvottu. Kun sovellus jatkossa lähettää pyyntöjä palvelimelle, niin pyyntöihin liitetään tämä eväste. Palvelin tarkistaa saapuvien pyyntöjen evästeet ja osaa niiden perusteella antaa pyyntöihin niitä vastaavan kontrolli- tai testiversion vastauksen. Evästeeseen voidaan tallentaa tieto käyttäjän ryhmästä usealla eri tavalla.

*What are cookies?, https://us.norton.com/internetsecurity-privacy-what-are-cookies.html, luettu 27.2.2020

Esimerkiksi Tang ja kumppanit kertovat tallentavansa käyttäjien evästeisiin juoksevan numeron, josta palvelin ottaa jakojäännöksen luvulla 1000 ja tämän tuloksen perusteella määritetään käyttäjän ryhmä (Tang et al., 2010). Yhteen kokeiluun saattaa tällöin esimerkiksi olla määrätty kaikki käyttäjät, joiden luvun jakojäännös on välillä 1-5 ja toiseen 6-10. Tällöin näihin kahteen kokeiluun pitäisi ohjautua sama määrä käyttäjä.

Evästeisiin tehtävään käyttäjille määritetyn variaation tallennukseen liittyy kuitenkin joitain erityispiirteitä, jotka kokeilujen tekijöiden on hyvä tiedostaa. Ensinnäkin evästeet eivät varsinaisesti yksilöi käyttäjää, sillä osaa laitteista ja niiden selaimista saattaa käyttää useampi käyttäjä. Tällaisia ovat muun muassa julkiset tietokoneet, kuten esimerkiksi kirjastojen ja Internet-kahviloiden tietokoneet sekä perheiden yhteiskäytössä olevat tietokoneet tai tabletit. Tällöin kokeilun näkökulmasta yksi käyttäjä onkin todellisuudessa useampi käyttäjä eikä kokeilun tulokset tämän kohteen osalta ole yhtä luotettavia, sillä nämä eri käyttäjät saattavat käyttäytyä varsin eri tavoilla. Lisäksi evästeet on helppo tyhjentää selaimen muistista ja monet käyttäjät tekevätkin tätä melko usein. Kolme kymmenestä käyttäjästä tyhjentää selaimensa evästeet kuukauden sisällä ja evästeiden keskimääräinen tyhjennystiheys on jopa neljä kertaa kuukaudessa*. Kun kokeilua tekevän sivuston evästeet poistetaan, niin yksi käyttäjä näkyy kokeilun kannalta kahtena erillisenä käyttäjänä. Nämä kokeilun tuloksia vääristävien käyttäjien aiheuttamat ongelmat voidaan kuitenkin jättää huomiotta mikäli kokeiluun osallistuvien käyttäjien määrä on riittävän suuri. Tällöin useamman käyttäjän tapaukset eivät ole kokeilun kannalta tilastollisesti merkittäviä.

Monien kokeilujen kohteena olevissa sovelluksissa käyttäjät voivat kirjautua sovellukseen tai sovelluksen käyttäminen saattaa jopa vaatia kirjautumisen. Tällaisiin käyttäjiin kohdistuvissa kokeiluissa käyttäjille määrätty ohjelmiston variaatio voidaan tallentaa selaimen evästeitä luotettavammin tietokantaan. Tällöin voidaan välttyä edellä mainituilta evästeisiin liittyviltä ongelmilta kokonaan. Kokeilun tekijöille kirjautuneista käyttäjistä on myös muita etuja, sillä heistä sillä heistä on usein saatavilla enemmän tietoa kuin käyttäjistä, jotka eivät ole kirjautuneet. Tällaisia tietoja voi olla esimerkiksi käyttäjän asuinpaikka, sukupuoli tai ikä. Joissain palveluissa käyttäjistä on saatavilla vielä paljon yksityiskohtaisempia tietoja, mikä mahdollistaa kokeilun kohdentamisen tietylle käyttäjäryhmälle. Esimerkiksi Linkedin-palveluun tehtävät kokeilut voidaan hyvin helposti rajata tietyn maan käyttäjiin (Xu et al., 2015).

*Cookie Deletion Rates and the Impact on Unique Visitor Counts, https://www.comscore.com/chi/Insights/Blog/Cookie-Deletion-Rates-and-the-Impact-on-Unique-Visitor-Counts, luettu 27.2.2020

Ennen kokeilun aloittamista voi olla hyödyllistä jakaa käyttäjiä testi- ja kontrolliryhmiin ja kerätä käyttäjistä samaa dataa kuin itse kokeilussakin on tarkoitus kerätä, mutta pitää molempien ryhmien ohjelmiston variaationa kontrolliversio (Tang et al., 2010). Tätä menetelmää kutsutaan myös A/A-testauseksi. Tällä voidaan varmistaa, että käyttäjien jakaminen testi- ja kontrolliversioihin toimii oikein ja että niihin ohjataan kokeilussa suunnitellut määrät käyttäjiä (Kohavi ja Longbotham, 2017). Lisäksi tällä menetelmällä voidaan varmistaa, että itse kokeilun tekeminen ei vaikuta kokeilussa mitattaviin arvoihin tai jos vaikuttaa, niin vaikutuksen täytyy olla sama sekä testi- että kontrolliversiossa tai muuten kokeilun tulokset eivät ole luotettavia. Vaikutus on myös otettava huomioon kokeilun tulosten analysoinnissa. Jos kokeilun tekeminen esimerkiksi hidastaa sovelluksen suorituskykyä jollain lailla, niin tämä tulee hyvin todennäköisesti näkymään myös kokeilun tuloksissa. Tällöin on tärkeää, että tämä suorituskyvyn heikentyminen ei kohdistu vain toiseen versioon. A/A-testausta voidaan tehdä myös kokeilun jälkeen.

Variaatioiden luomisen ja niihin käyttäjien jakamisen jälkeen on olennaista kerätä näiden käyttäjien toimista dataa kattavasti ja tarkasti, jotta itse kokeilun onnistumista voidaan arvioida. Seuraavassa kappaleessa käsitellään datan keräystä, kerätyn datan käsittelyä ja lopulta datan analysointia.

4.3 Datan keräys, käsittely ja analysointi

Tässä kappaleessa käsitellään käyttäjädatan keräystä. Ensin tarkastellaan ohjelmistokehityksessä tehtävää lokitusta yleisesti ja miten sitä pitäisi muuttaa, jotta se soveltuisi paremmin jatkuvaan kokeilemiseen. Lisäksi käsitellään lyhyesti muita datan keräysmenetelmiä. Tämän jälkeen käsitellään sitä miten kerättyä dataa käsitellään ja muunnellaan, jotta sitä olisi helpompi tarkastella. Lopuksi käydään läpi käsitellyn datan analysointia ja siihen liittyviä toimenpiteitä.

4.3.1 Yleiskatsaus lokitukseen

Lokitus on yleinen ohjelmistokehityksen tapa ja useimmat ohjelmistot lokittavatkin toimintaansa jollakin tavalla (Yuan et al., 2012). Tämä saattaa olla esimerkiksi ohjelmiston toiminnan analysointia varten tehtyä lokitusta tai ongelmatilanteiden korjausta varten tehtyä lokitusta. Jälkimmäisessä lokimerkintöjä tehdään virhetilanteista ja

odottamattomista tapahtumista, tai testausta varten tehtyä lokitusta, jonka tarkoitus on auttaa jonkin toiminnallisuuden eheyden varmistamisessa (Pecchia et al., 2015). Erityisesti tuotantojärjestelmien ongelmien korjauksessa hyvin tehty lokitus voi auttaa kiireellisen ongelman paikantamisessa ja korjaamisessa merkittävästi. Lokitus voi myös auttaa kriittisten järjestelmien ongelmien ennakkoimisessa, jonka takia lokituksen tärkeys korostuukin näiden järjestelmien kohdalla (Pecchia et al., 2015).

Usein erityyppisiä ja eri tarkoituksiin tehtyjä lokimerkintöjä varten käytetään erilaisia lokitustasoja. Kuvasta 4.5 nähdään esimerkkejä tyypillisistä eritasoisista lokimerkinnöistä.

```
elog (FATAL, "out of memory"); /* PostgreSQL */
ap_log_error(ERR, "could no open charset \conversion config
   file %s", confname); /* Apache */
logit (INFO, "Authentication refused %s", line); /* OpenSSH */
elog(DEBUG1, "archived transaction log file %s", xlog); /*
   PostgreSQL */
```

Kuva 4.5: Lokitusesimerkkejä avoimen lähdekoodin projekteista (Yuan et al., 2012)

Lokituksen helpottamiseksi ja yhtenäistämiseksi on saatavilla useita ohjelmistokehyksiä. Tällaisia ovat esimerkiksi Apache Log4j 2* ja SLF4J† Java-ohjelmointikielelle sekä Apache log4net‡ .NET ohjelmistokehykselle. Ohjelmistokehykset auttavat lokiviestien yhtenäistämisessä, keräämisessä, parsimisessa sekä filtteröinnissä. Kuva 4.6 antaa esimerkkejä Java-ohjelmointikielellä ja Apache Log4j 2 -ohjelmistokehyksellä tehdystä lokituksesta. Tästä syntyy lokimerkintöjä, jotka sisältävät lokimerkinnän tarkan aikaleiman, lokimerkinnän tason, lokimerkinnän tehneen luokan ja rivinumeron sekä itse lokimerkinnän viestin. Kuvassa 4.7 on esitetty nämä lokimerkinnät.

Hyödyllisyydestään huolimatta lokituksen ohjelmistokehykset eivät kuitenkaan auta päättämään sitä, mitkä tapahtumat pitäisi lokittaa, vaan tämä jää ohjelmistokehittäjien itsensä tehtäväksi (Pecchia et al., 2015). Tässä on useita eri osa-alueita, jotka jäävät jokaisen ohjelmistokehittäjän pohdittavaksi, kuten lokituksen tehtävä, lokituksen sijainti koodissa sekä itse lokimerkinnän viesti. Tästä syystä lokituskäytännöt eivät

*Apache Log4j 2, https://logging.apache.org/log4j/2.x/, luettu 31.1.2020
†SLF4J, http://www.slf4j.org/, luettu 31.1.2020
‡Apache log4net, https://logging.apache.org/log4net/, luettu 31.1.2020

```
...
Logger logger = Logger.getLogger(Example.class);
logger.debug("Tämä on debug -tason lokiviesti");
logger.info("Tämä on info -tason lokiviesti");
logger.warn("Tämä on warn -tason lokiviesti");
logger.error("Tämä on error -tason lokiviesti");
logger.fatal("Tämä on fatal -tason lokiviesti");
...
```

Kuva 4.6: Apache Log4j 2 -ohjelmistokehyksen lokitusesimerkkejä

```
2020-01-31 20:52:39 DEBUG Example:19 - Tämä on debug -tason
    lokiviesti
2020-01-31 20:52:39 INFO  Example:20 - Tämä on info -tason
    lokiviesti
2020-01-31 20:52:39 WARN  Example:21 - Tämä on warn -tason
    lokiviesti
2020-01-31 20:52:39 ERROR Example:22 - Tämä on error -tason
    lokiviesti
2020-01-31 20:52:39 FATAL Example:23 - Tämä on fatal -tason
    lokiviesti
```

Kuva 4.7: Apache Log4j 2 -ohjelmistokehyksen lokitusmerkintöjä

välttämättä ole yhtenäisiä edes saman tiimin sisällä, sillä lokituksen toteutus riippuu paljon ohjelmistokehittäjien aiemmasta kokemuksesta sekä subjektiivisista näkemyksistä. Myös Yuan ja kumppanit (Yuan et al., 2012) havaitsivat tämän tutkiessaan avoimen lähdekoodin projekteja. Tämän seurauksena lokituskoodin laatu on usein huonompaa kuin muun ohjelmakoodin.

Tämän lisäksi tyypillisesti lokitus keskittyy enimmäkseen virhetilanteisiin ja testaukseen, jolloin se ei myöskään kerro siitä, miten käyttäjät käyttävät ohjelmistoa (Fabijan, Dmitriev, McFarland et al., 2018). Tällaisen lokituksen datasta on todennäköisesti hyvin hankalaa tehdä luotettavia päätelmiä ja analyysejä ohjelmiston käytöstä. Jatkuvaa kokeilua varten onkin siirryttävä tavallista ohjelmistokehitystä kattavampaan ja

järjestelmällisempään lokitukseen.

4.3.2 Datan keräys jatkuvassa kokeilemisessa

Lokitus nousee erityisen keskeiseen rooliin jatkuvassa kokeilemisessa. Yleisesti ottaen lokituskoodi ei kuitenkaan ole monille ohjelmistokehittäjille yhtä tärkeässä asemassa kuin ohjelmiston toiminnasta vastaava koodi. Jatkuvaan kokeilemiseen siirtyessä lokituskoodin laatuun on ryhdyttävä kiinnittämään yhtä paljon huomiota kuin muun koodin laatuun (Gupta et al., 2018). Jatkuvaa kokeilemista varten tehdyn lokituksen keskeisenä tavoitteena on, että kuka tahansa ohjelmistokehittäjä tai datatieteilijä voi lukea lokiin kerättyä dataa ja saada hyvän kuvan lokitetuista tapahtumista ilman tietoa kokeilun kohteena olevasta tuotteesta (Fabijan, Dmitriev, Olsson et al., 2017b). Lokimerkintöjen pohjalta pitää myös voida yksiselitteisesti toistaa kenen tahansa kokeilussa mukana olevan käyttäjän tekemät toiminnot (Gupta et al., 2018).

Näiden vaatimusten täyttäminen edellyttää, että lokitusta tehdään järjestelmän jokaisessa osassa (Kohavi, Deng, Frasca et al., 2013). Yksittäisen käyttäjän kohdalla lokitustapahtumat alkavat siitä, kun käyttäjä määrätään järjestelmän toimesta johonkin kokeiluun. Tapahtumasta tehdään merkintä riippumatta siitä määrätäänkö käyttäjä testi- vai kontrolliryhmään. Merkintään kirjataan ainakin kokeilun tunniste ja aikaleima sekä mahdollisesti muita oleellisia tietoja (Xu et al., 2015). Tämän jälkeen lokitusta tehdään itse kokeilun kohteena olevassa järjestelmässä kaikista käyttäjän toiminnoista. Muutamia esimerkkejä lokitettavista toiminnoista tai tapahtumista ovat klikkaukset, mobiililaitteella tehtävät pyyhkäisyt, vuorovaikutus tuotteen kanssa, sovelluksen latausajat ja vuorovaikutus tiedostojen kanssa (Fabijan, Dmitriev, Olsson et al., 2017b). Suurimmassa osassa ohjelmistoja lokitusta on siis tehtävä sekä asiakas- että palvelinohjelmistossa (Gupta et al., 2018).

Asiakasohjelmistoissa tehtävässä lokituksessa on usein otettava huomioon asiakasohjelmiston rajoitukset. Data on lähetettävä asiakasohjelmistosta lokitiedot tallentavalle palvelimelle, mutta tehdäänkö datan lähetys reaaliaikaisesti heti lokitettavan toiminnon tapahduttua vai lähetetäänkö tietyin väliajoin isompi määrä dataa, joka on väliaikaisesti tallennettu asiakasohjelmistoon. Molemmissa lähestymistavoissa on otettava huomioon tilanteet, joissa asiakasohjelmisto ei saa yhteyttä palvelimelle. Tällaisia voi olla esimerkiksi tilanteet, joissa Internet-yhteyttä ei väliaikaisesti ole saatavilla tai lokituspalvelin ei jostain syystä ota vastaan dataa.

Selainsovelluksessa dataa saattaa kadota, jos sovelluksella kestää kauan latautua ja käyttäjä sulkee sivun ennen kuin se latautuu. Tällöin tietoa tästä epäonnistuneesta sivulatauksesta ei ehdi tallentua. Myös pidempi aika ilman Internet-yhteyttä voi johtaa selaimen välimuistin täyttymiseen ja sen myötä datan katoamiseen (Gupta et al., 2018). Monissa sulautettujen järjestelmien asiakasohjelmistoissa ei ole ollenkaan paikallista tallennustilaa käytettävissä, jolloin lokitettavista tapahtumista on lähetettävä data lokituspalvelimelle heti niiden tapahduttua tai data menetetään (Bosch ja Eklund, 2012). Kaikki tällaiset poikkeustilanteet on otettava huomioon asiakasohjelmistoista dataa kerättäessä. Tämän takia on myös aina parempi kerätä data palvelimen päässä kuin asiakasohjelmistossa, jos vain mahdollista (Gupta et al., 2018). Tällöin vältytään datan menettämisen riskiltä ja lisäksi datan saamisessa ei ole samanlaista viivettä kuin kerättäessä sitä asiakasohjelmistosta.

Lokitapahtumien nimeäminen on myös olennainen osa lokitusta. Kuten aiemmin mainittiin, niin lokitusta on usein tehtävä monissa eri osissa järjestelmää. Useimmissa ohjelmistoissa vähintäänkin kahdessa eri osassa eli asiakas- ja palvelinohjelmistossa. Nämä eri osat saattavat olla toisistaan täysin erillisiä, jolloin ne ovat myös mahdollisesti eri ohjelmistokehittäjien tai ohjelmistokehitystiimien kehittämiä ja ylläpitämiä. Tästä syystä nimeämiskäytäntöjen pitäisi olla yhdenmukaista järjestelmän eri osien välillä, jotta lokituksen ymmärtäminen ja tulkitseminen olisi kaikille mahdollisimman vaivatonta. Nimeämiskäytäntöjen tulisi olla yhdenmukaisia jopa yrityksen sisällä kokonaan erillisissä tuotteissa (Fabijan, Dmitriev, Olsson et al., 2017b).

Nimeäminen voi myös olla tärkeää lokitukseen käytetyn tilan kannalta, sillä kerätyn datan määrä voi kasvaa isoissa palveluissa valtavaksi. Esimerkiksi Linkedin-palvelusta lokitetaan neljä teratavua käyttäjien toimista ja kuusi teratavua kokeiluihin määrätyistä käyttäjistä kertovaa dataa joka päivä (Xu et al., 2015). Varsinkin näin suurien datamäärien kohdalla lokitapahtumien pitäminen mahdollisimman pieninä on tärkeää sekä tilan säästämisen että verkkoliikenteen alhaisempana pitämisen kannalta.

Selainsovellusten kohdalla on yleistä, että iso osa liikenteestä on erilaisia botteja. Boteilla tarkoitetaan tässä ohjelmistoja, jotka vierailevat automatisoidusti Internet-sivustoilla. Botteja on monia erilaisia ja moniin tarkoituksiin, kuten esimerkiksi hakukoneita varten indeksointia tekeviä botteja, sivustoilta dataa keräämiä botteja sekä monia haavoittuvuuksia etsiviä botteja. Vuonna 2016 arvioitiin, että yli puolet selainsovellusten liikenteestä tuli boteilta*. Myös Microsoftin Bing-hakukoneen kehittäjät raportoivat

*Bot Traffic Report 2016, https://www.imperva.com/blog/bot-traffic-report-2016/, luettu

samankaltaisia lukuja bottiliikenteen määrästä*.

Kokeiluiden kannalta boteilta tuleva liikenne saattaa vaikuttaa kokeilun tuloksiin, jonka takia tämän liikenteen määrää on hyödyllistä yrittää minimoida mahdollisimman paljon (Kohavi, Longbotham et al., 2009; Rodden et al., 2010). Niin kutsutut hyväntahtoiset botit, eli esimerkiksi hakukoneiden indeksointia tai muuta datan keräystä tekevät botit, voidaan yleensä suodattaa melko helposti yksinkertaisella robots.txt -tiedostolla, joka kertaa botille millä sivuston poluille sen pitäisi vierailla ja millä taas ei. Pahantahtoisia haavoittuvuuksia etsiviä botteja voi olla hankalampi suodattaa. Yksinkertaisia botteja voidaan yrittää tunnistaa esimerkiksi käyttäjäagentin (eng. user agent) tai IP-osoitteen perusteella, mutta nykypäivänä monet pahantahtoiset botit osaavat kiertää nämä tunnistamismenetelmät (Kohavi ja Longbotham, 2017). Jo aiemmin mainittu A/A-testaus voi olla avuksi tässäkin tapauksessa, sillä A/A-testistä voidaan nähdä bottien vaikutusta kun kontrolliversiota testataan itseään vasten. A/A-testaus myös varmistaa datan keräyksen toimivuuden yleisesti. A/A-testissä kerätään dataa käyttäjien toimista samalla lailla kuin tavallisesta kokeilustakin kerättäisiin dataa. Testi- ja kontrolliversioista kerättyä dataa verrataan A/A-testin jälkeen keskenään ja mikäli niistä kerätty data on samanlaista sekä kerätyn datan määrä molemmista versioista on oikeassa suhteessa, niin voidaan olla varmoja, että datan keräys toimii oikein.

Lokituksen kautta tehtävä datan keräys ei ole ainoa tapa kerätä dataa käyttäjien toimista. Käyttäjillä voidaan myös teettää erilaisia kyselyitä, joilla voidaan selvittää käyttäjien käyttökokemuksia jostakin tietystä toiminnallisuudesta tai laajemmasta kokonaisuudesta. Kyselyitä voidaan tehdä myös ennen kuin toiminnallisuutta, johon kysely kohdistuu, on vielä toteutettu. Käyttäjiltä voidaan esimerkiksi selvittää kyselyn ja siihen liitettyjen luonnosten avulla, että millainen käyttöliittymä vaikuttaisi heidän mielestään parhaalta ennen kuin sen toteutus on aloitettu (Tullis ja Albert, 2013). Tämän lähestymistavan etu käyttäjillä tehtäviin kokeiluihin nähden on sen toteuttamisen helppous ja kustannustehokkuus, sillä luonnoksen ja kyselyn tekeminen on todennäköisesti merkittävästi helpompaa tehdä kuin varsinaisen käyttöliittymän toteutus.

Kyselyiden teettämiseen on myös olemassa useita valmiita työkaluja, joiden avulla kyselyitä voidaan teettää pienellä vaivalla. Tällaisia työkaluja ovat esimerkiksi Survey-

24.2.2020

*The surprising Power of Online Experiments, https://hbr.org/2017/09/the-surprising-power-of-online-experiments, luettu 24.2.2020

Monkey* ja SurveyGizmo†. On kuitenkin pidettävä mielessä, että kokeiluilla kerätty data on paljon laadukkaampaa kuin kyselyillä kerätty data, sillä kokeilut kertovat tarkalleen käyttäjien toiminnasta, kun taas kyselyiden tuloksiin vaikuttavat monet ulkopuoliset tekijät (Kohavi, Deng, Frasca et al., 2013). Tästä syystä kyselyitä pitäisikin pitää vain kokeiluita täydentävänä tapana kerätä dataa käyttäjistä eikä yrittää korvata kokeiluita niillä.

4.3.3 Datan jatkokäsittely

Datan keräämisen jälkeen sitä on muokattava ja yhdisteltävä eri lähteistä saadun datan kanssa erilaisilla tavoilla. Tämän tarkoituksena on tehdä datasta paremmin ymmärrettävää ja helposti käsiteltävää niille, jotka tekevät sen pohjalta analyysejä ja päätelmiä (Gupta et al., 2018). Datan yhdistämisen ensimmäinen vaihe on yhdistää yksittäisistä käyttäjien toiminnoista koostuva data siihen dataan, joka kertoo mihin kokeiluun käyttäjä kuuluu ja milloin hänet on siihen liitetty (Xu et al., 2015). Mikäli kokeilun kohteena olleeseen sovellukseen voi kirjautua, niin kirjautuneiden käyttäjien käyttäjätilien tietoja on myös usein hyödyllistä liittää tässä vaiheessa muuhun dataan.

Usein käyttäjien tekemät toiminnot on syytä jakaa erilaisiin kategorioihin, jotta niiden yhdisteleminen ja niistä erilaisten tulosten johtaminen olisi mahdollisimman helppoa (Fabijan, Dmitriev, Olsson et al., 2017b). Fabijan ja kumppanit suosittelevat jakamaan erityyppiset toiminnot, joita he kutsuvat signaaleiksi, omiin kategorioihinsa. Heidän luokituksessaan on kolmenlaisia signaaleja:

- Toimintasignaalit, sisältävät muun muassa sivulataukset ja klikkaukset
- Aikasignaalit, sisältävät muun muassa sessioiden kestot ja sivulatausten kestot
- Arvosignaalit, sisältävät muun muassa ostotapahtumat ja mainosklikkaukset

Yhdistelemällä erilaisia signaaleja ja aikamääreitä voidaan laskea erilaisia metriikoita. Fabijan ja kumppanit (Fabijan, Dmitriev, Olsson et al., 2017b) määrittelevät metriikan funktioksi, joka ottaa syötteenä yhden tai useamman signaalin ja antaa tulokseksi luvun jotakin yksikköä kohden. Esimerkiksi selainsovelluksille hyvin tärkeä ja yleisesti laskettava metriikka on klikkaussuhde (eng. click-through rate), joka kertoo kuinka

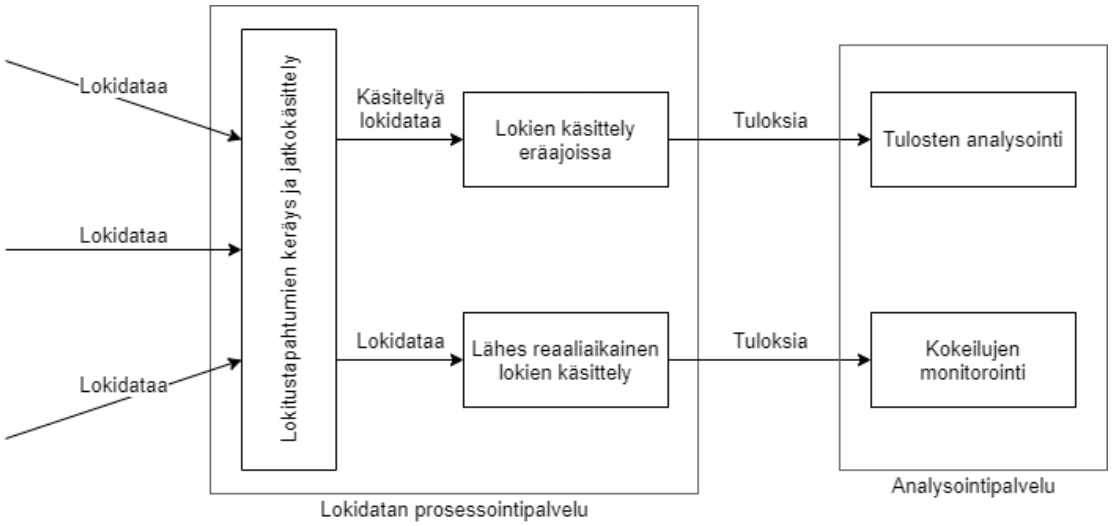
*SurveyMonkey, https://fi.surveymonkey.com/, luettu 24.4.2020
†SurveyGizmo, https://www.surveygizmo.com/, luettu 24.4.2020

suuri osa jonkin sivun ladanneista käyttäjistä on klikannut jotakin tiettyä sivulla sijaitsevaa linkkiä tai nappia. Klikkaussuhde lasketaan jakamalla linkkiä tai nappia klikanneiden käyttäjien lukumäärä kaikkien sivun ladanneiden käyttäjien lukumäärällä. Toinen monille selainsovelluksille keskeinen metriikka on sivun suorituskyvystä kertova sivulatauksen metriikka. Tämä saadaan tallentamalla jokaiseen sivulataukseen kulunut aika ja laskemalla tästä keskiarvo kaikkien käyttäjien kesken. Signaalien dataa käsiteltäessä ja metriikoita laskiessa on tärkeää kiinnittää huomiota siihen, että mitään dataa ei saisi näiden seurauksena kadota (Gupta et al., 2018).

Klikkaussuhde on hyvä esimerkki metriikasta myös siltä kannalta, miten asiakasohjelmistossa kerättävä data on yhdistettävä palvelimella kerättyyn dataan. Klikkaussuhteen laskemisessa käytettävä tieto sivulatauksesta tallennetaan tässä tapauksessa palvelimella, kun käyttäjä navigoi kokeilun kohteena olevalle sivulla ja tieto siitä kun käyttäjä klikkaa sivulla sijaitsevaa nappia tai linkkiä tallennetaan alunperin asiakasohjelmistossa (Gupta et al., 2018). Klikkaussuhde ja keskimääräisen sivulatauksen kesto ovat kuitenkin vain kaksi esimerkkiä isosta joukosta laskettavia metriikoita. Yhteistä kaikille metriikoille kuitenkin on, että niiden tulisi olla mahdollisimman helposti muokattavia, jotta esimerkiksi uuden metriikan lisääminen onnistuu vaivatta (Gupta et al., 2018).

Kuva 4.8 havainnollistaa esimerkkitapausta järjestelmän osista, joissa tehdään lokidatan keräystä, jatkokäsittelyä ja analysointia kokeiluja varten ja miten nämä järjestelmän osat liittyvät toisiinsa. Järjestelmään tulee jatkuvasti lokidataa useasta eri lähteestä ja se käsitellään yhdessä osassa järjestelmää. On tärkeää, että tämä dataa käsittelevä ja yhdistelevä järjestelmän osa pystyy käsittelemään viiveellä tulevan datan oikein, sillä kuten aiemmin mainittiin, niin asiakasohjelmistoista kerättyä dataa ei aina ole mahdollista lähettää palvelimelle heti sen keräämisen jälkeen (Gupta et al., 2018). Tässä järjestelmän osassa dataan tehdään tarvittaessa muutoksia ja sitä yhdistellään mahdollisesti muuhun dataan. Jos dataa kerätään isoja määriä, niin erilaisten metriikoiden laskeminen siitä voi olla merkittävän paljon resursseja ja aikaa vievää. Metriikoiden laskemista tehdään tämän takia usein isommissa eräajoissa tietyin aikavälein. Lopuksi metriikkadata siirretään vielä toiseen järjestelmään, joka on tarkoitettu tulosten analysointiin. Tämä järjestelmä saattaa esimerkiksi helpottaa eri kokeilujen tulosten analysoimista tarjoamalla erilaisia datan analysointi- tai vertailutyökaluja.

Käyttäjiä koskevaa dataa käsiteltäessä on otettava huomioon datan anonymisointi (Fabijan, Dmitriev, McFarland et al., 2018). Tästä on tullut erityisen tärkeää vuonna 2016



Kuva 4.8: Lokitusjärjestelmän osia (Gupta et al., 2018)

annetun EU:n yleisen tietosuoja-asetuksen (eng. General Data Protection Regulation, GDPR) myötä*. Tämän asetuksen myötä yrityksille voidaan määrätä merkittäviä sakkoja, mikäli yrityksen keräämää henkilötietoja sisältävää dataa joutuu vääriin käsiin (Tankard, 2016). Henkilötietoihin katsotaan kuuluvan myös esimerkiksi IP-osoitteet ja selaimen evästeet. Lisäksi yleinen tietosuoja-asetus antaa käyttäjille mahdollisuuden vaatia heistä kerätyn henkilötietoja sisältävän datan poistamista. Tällaisen vaatimuksen toteuttaminen voi olla erittäin työlästä, mikäli käyttäjiä yksilöiviä tietoja ei ole anonymisoitu datan jatkokäsittelyvaiheessa.

Datan käsittelyn lopputuote koostuu usein erilaisista metriikoista, joista on lisäksi yleensä koostettu jonkinlainen yhteenveto. Tämän yhteenvedon tarkoitus on osoittaa yksiselitteisesti menestyikö testiversio kokeilussa paremmin kuin kontrolliversio. Erilaiset lasketut metriikat mahdollistavat kuitenkin kokeilun syvällisemmän analysoinnin. Tämä on erityisen tärkeää silloin, kun testiversio todetaan kokeilun perusteella kontrolliversiota huonommaksi. Tällöin metriikoiden tulkitsemisesta voidaan oppia uusia asioita käyttäjistä ja käyttäjien toiminnasta, joka on avuksi ohjelmiston tulevaa kehitystä ja seuraavia kokeiluita suunniteltaessa.

4.3.4 Datan analysointi

Datan analysointivaiheessa tehdään varsinaiset päätelmät siitä mikä ohjelmiston variaatio suoriutui kokeilussa parhaiten. Lisäksi analysointivaiheen tehtävä on korjata

*General Data Protection Regulation, https://gdpr-info.eu/, luettu 20.4.2020

aiempia oletuksia käyttäjien käyttäytymisestä ja haluista liiketoiminnan edistämiseksi (Mattos et al., 2018). Tämän analyysin jälkeen pitäisi olla selvää otetaanko kokeilun testiversio käyttöön vai pysytäänkö kontrolliversiossa. Analysoinnin oikeellisuus perustuu siihen, että sitä edeltävät vaiheet, datan keräys ja datan jatkokäsittely, on tehty oikein ja riittävästä määrästä käyttäjiä. Se, miten työläs tämä vaihe on voi vaihdella paljonkin riippuen siitä, miten syvällisesti tehdyn kokeilun tuloksia on tarpeen analysoida, sekä siitä millaisia tuloksia datan käsittelystä saatiin.

Analysointivaiheessa kokeilun tekijät ja muut kokeiluun liittyvät sidosryhmät keskustelevat kokeilun tuloksista ja päättävät yhdessä oliko kokeiltu muutos onnistunut ja se pitäisi ottaa käyttöön, vaatiiko tämän päättäminen jonkin uuden kokeilun vai pitäisikö muutoksesta luopua kokonaan (Tang et al., 2010). On myös melko yleistä, että tässä vaiheessa päätetään muokata tai tarkentaa kokeilua ja suorittaa se uudestaan näillä muutoksilla. Kokeileminen voi siis usein olla iteratiivinen prosessi myös yksittäisen kokeilun skaalassa. Näiden analysointivaiheen keskustelujen tulokset pitäisi dokumentoida huolellisesti, jotta myös tämän vaiheen tulokset ovat tallessa ja jälkikäteen saatavilla, kuten kokeilun aiempienkin vaiheiden tulokset (Tang et al., 2010). Keskustelut ovat myös hyödyllisiä kokeiluiden tekemiseen liittyvän osaamisen kehittämisessä organisaation sisällä, sillä vähemmän kokemusta omaavat henkilöt voivat saada niistä paljon uutta osaamista kokeiluiden tulosten tulkintaan ja analysointiin.

Analysointivaiheessa tulisi varmistaa, että kokeilun tulokset ovat luotettavia eikä kokeilussa tai datan keräyksessä ole tapahtunut virheitä. Tässä auttaa, että kokeilun kohteena oleva ohjelmisto ja kokeilualusta ja kaikki näihin liittyvät järjestelmän osat ovat kokeilun tekijöille tuttuja (Tang et al., 2010). Jo aiemmin mainitut A/A-testit ovat myös erittäin hyvä keino varmistua kokeilun tekemisen ja datan keräyksen toimivuudesta (Kohavi ja Longbotham, 2017). Turhien kokeiluiden välttämiseksi A/A-testaus on usein hyödyllisempää tehdä ennen kokeilua, mutta jos analysointivaiheessa kokeilun tulosten oikeellisuudesta on epäilyjä, niin A/A-testi voidaan suorittaa myös itse kokeilun jälkeen.

Tarkasteltavat metriikat ovat jatkuvan kokeilemisen analysointivaiheen keskiössä, jonka takia niiden suunnittelemiseen ja määrittelemiseen on syytä käyttää erityistä huolellisuutta. Metriikoiden tärkein tehtävä on antaa kokonaisvaltainen ja yksiselitteinen kuva siitä, mitä kokeilussa on tapahtunut (Tang et al., 2010). Kokonaisvaltainen tarkoittaa tässä sitä, että analysointivaiheessa on syytä tarkastella useaa metriikkaa, jotta kokeilun vaikutuksista saadaan tarpeeksi hyvä kuva (Rodden et al., 2010). Seuraavaksi

tarkastellaan kahden eri organisaation lähestymistapoja metriikoiden luokitteluun ja käyttöön.

Rodden ja kumppanit (Rodden et al., 2010) huomioivat, että yleisimmin käytetyt metriikat ovat sivun näyttökerrat, käytettävyysaika, viive, aktiivisten käyttäjien lukumäärä seitsemän päivän aikana sekä ansainnat. Nämä ovat tärkeitä metriikoita käyttäjäkokemuksen mittauksessa, mutta niiden ongelma on, että ne mittaavat käyttäjäkokemusta epäsuorasti, jonka takia ne ovat ongelmallisia mitattaessa käyttöliittymään kohdistuvia muutoksia. Rodden ja kumppanit (Rodden et al., 2010) ehdottavatkin näiden metriikoiden sijaan heidän Googlella käyttämiään käyttäjäkeskeisempiä metriikoita, jotka he ovat jakaneet viiteen luokkaan:

- Onnellisuus. Sisältää muun muassa käyttäjän tyytyväisyyttä ja sovelluksen mielletyn käytettävyyden helppouden metriikoita.
- Sitoutuminen. Sisältää käyttäjien käyttöasteita mittaavia metriikoita, eli kuinka paljon käyttäjät käyttävät sovellusta.
- Omaksuminen. Sisältää uusiin käyttäjiin keskittyviä metriikoita.
- Pysyvyys. Sisältää käyttäjien jatkuvaa käyttöä pidemmällä aikavälillä mittaavia metriikoita.
- Tehtävässä onnistuminen. Sisältää käyttäjien sovelluksen käytön tehokkuutta ja onnistumista mittaavia metriikoita.

Tämän luokittelun tarkoitus on helpottaa sopivien metriikoiden valitsemista. Suurimmassa osassa kokeiluista jokaisesta luokasta tulisi valita ainakin yksi metriikka. Niissä tapauksissa, joissa jokin luokka ei ole kokeilun kannalta olennainen, niin luokittelu pakottaa kuitenkin kokeilun tekijät tiedostamaan ja keskustelemaan siitä miksi tämän luokan metriikat eivät ole olennaisia.

Xu ja kumppanit havainnollistavat (Xu et al., 2015) Linkedin-palvelun kokeiluiden metriikoita, jotka ovat jaettuina kolmeen eri tasoon. Korkeimman tason metriikat ovat koko yrityksen laajuisia, joita alempana ovat tuotekohtaiset metriikat ja alimpana ovat toiminnallisuuskohtaiset metriikat. Korkeimman tason koko yrityksen laajuisten metriikoiden ylläpidosta ja päivittämisestä vastaa oma keskitetty tiiminsä, kun taas alemman tason metriikoista ovat vastuussa tuotekehitystiimit. Tällä metriikoiden jakamisella tasoihin pyritään siihen, että yrityksen kannalta tärkeimmät metriikat otetaan kaikissa

kokeiluissa huomioon ja ne ovat aina ajan tasalla, mutta alemman tason metriikat tarjoavat kuitenkin kokeiluiden tekijöille mahdollisuuden syvällisempään analysointiin (Xu et al., 2015).

On huomioitava, että varsinkin monilla pienemmillä yrityksillä ei välttämättä ole yhtä paljon resursseja metriikoiden ylläpitoon ja päivittämiseen. Linkedin-palvelun esimerkki kuitenkin osoittaa hyvin metriikoiden keskeisen aseman kokeiluissa sekä metriikoiden luokituksesta saatavan edun.

Kuten aiemmin mainittiin, niin kokeilussa mukana olevien käyttäjien määrä on olennainen kokeilun tulosten luotettavuuden kannalta. Jos kokeilussa on liian alhainen määrä käyttäjiä, niin kokeilun tilastollinen voimakkuus on alhainen. Tilastollinen voimakkuus tarkoittaa todennäköisyyttä, jolla kokeilulla havaitaan kokeilun kannalta merkityksellinen muutos metriikoissa (Kohavi, Longbotham et al., 2009). Kokeiluun tarvittava käyttäjämäärä määräytyy testiversioiden lukumäärästä sekä kokeilun kestosta.

Tulosten analysoinnissa tulee myös ottaa huomioon viikonpäivien vaikutukset. Tämä johtuu siitä, että käyttäjien käyttäytyminen on monien sovellusten kohdalla merkittävän erilaista arkipäivinä ja viikonloppuna. Tästä syystä kokeiluiden keston tulisi aina olla kokonaisia viikkoja, jotta esimerkiksi viitenä arkipäivänä ja neljänä viikonlopun päivänä ajettu testi ei anna väärää tulosta (Kohavi, Longbotham et al., 2009). Viikonloppujen lisäksi on otettava huomioon muut pyhäpäivät ja jopa lomakaudet.

4.3.5 Reaaliaikainen monitorointi

Monissa järjestelmissä tehdään reaaliaikaista monitorointia, vaikka niissä ei tehtäisi minkäänlaisia kokeiluita. Monitoroinnin avulla voidaan havaita ongelmia järjestelmässä nopeasti, jolloin niihin voidaan usein puuttua ennen kuin ne aiheuttavat käyttäjille merkittävää haittaa. Monitoroinnin tarve kuitenkin kasvaa merkittävästi, mikäli järjestelmässä tehdään kokeiluita (Schermann, Cito, Leitner et al., 2019). On esimerkiksi melko yleistä, että jollakin kokeilulla on varsin negatiivinen vaikutus käyttökokemukseen. Tällaisissa tapauksissa kokeilu halutaan keskeyttää jo hyvin aikaisessa vaiheessa (Fabijan, Dmitriev, Olsson et al., 2017b). Ongelman nopeaan huomaamiseen kokeilusta kerättävän datan seurannan täytyy kuitenkin olla reaaliaikaista. Tästä syystä monet jatkuvan kokeilemisen laajasti käyttöön ottaneet organisaatiot ovatkin toteuttaneet myös kattavaa järjestelmien ja kokeiluiden reaaliaikaista monitorointia (Tang et al., 2010).

Kokeiluiden reaaliaikasen valvonnan toteuttamiseen on useita lähestymistapoja. Yh-

teistä näille kuitenkin on, että käyttäjistä kerättyä lokidataa käsitellään jatkuvasti pieniä määriä lähes reaaliaikasesti (Gupta et al., 2018). Tätä tehdään erillisenä käsittelynä laajemmista datan käsittelyn eräajoista, kuten kuva 4.8 havainnollistaa. Käsitellyn datan tuloksia verrataan monitorointijärjestelmään asetettuihin raja-arvoihin. Mikäli raja-arvot ylittyvät, niin järjestelmän tulisi lähettää jonkinlaisia automaattisia hälytyksiä, jotta ongelmallinen kokeilu voidaan tarvittaessa keskeyttää nopeasti (Kohavi, Deng, Frasca et al., 2013). Nämä automaattiset hälytykset voivat olla esimerkiksi kokeilun vastuuhenkilöille lähetettävät sähköpostit tai muunlaiset ilmoitukset.

Usein monitorointijärjestelmä sisältää monia erilaisia tarkkailtavia raja-arvoja, jotka aiheuttavat automaattisia hälytyksiä. Selainpohjaisessa sovelluksessa tarkkailtavia arvoja voivat olla esimerkiksi käyttäjien sessioiden pituudet tai käyttäjien konversio, eli kuinka moni käyttäjä päätyy tilaamaan jotakin tuotetta, tai mainosklikkausten määrä.

Käsitellyn datan valvonnan lisäksi on tärkeää valvoa kerättävän datan laatua (Gupta et al., 2018). Tarkoituksenmukaisia ja tahattomia muutoksia datan käsittelyyn tehdään jatkuvasti ja nämä muutokset voivat vaikuttaa kokeilun metriikoiden oikeellisuuteen. Mikäli nämä muutokset tapahtuvat kokeilun tekijöiden huomaamatta, niin kokeilun tulosten analysoinnissa saatetaan päätyä väärään lopputulokseen. Datan laadun valvonnassa seurattavia asioita voivat olla esimerkiksi saapuvan datan määrä, poikkeavien havaintojen osuus ja tyhjät arvot.

Tähän asti tässä tutkielmassa on käsitelty monia jatkuvan kokeilemisen toteuttamisen kannalta tärkeitä osa-alueita. Näiden lisäksi jatkuvan kokeilemisen kannalta tärkeää on se, mihin kokeileminen kohdistetaan, sillä kokeilun kohde ja laajuus voi vaihdella paljon. Seuraavassa kappaleessa käydään näitä aiheita tarkemmin läpi.

4.4 Kokeilun kohde

Tässä kappaleessa tarkastellaan mihin kaikkeen jatkuvan kokeilemisen kokeilut voivat kohdistua. Lisäksi arvioidaan sitä minkä suuruisiin kokonaisuuksiin kokeiluiden pitäisi kohdistua. Onko esimerkiksi sovelluksen käyttöliittymää uudistettaessa parempi kokeilla pieniä muutoksia yksi kerrallaan vai koko käyttöliittymän uudistamista kerralla?

Jatkuvaa kokeilemista käsittelevässä kirjallisuudessa keskitytään usein eniten uusien toiminnallisuuksien kokeiluihin. Nämä uusien toiminnallisuuksien kokeilut kohdistuvat pääasiassa käyttöliittymiin, mutta myös muun toimintalogiikan, kuten algoritmien

testaus on yleistä (Tang et al., 2010). Schermann ja kumppanit (Schermann, Cito ja Leitner, 2018) kuitenkin huomioivat tekemänsä kyselyn perusteella, että vanhan toiminnallisuuden toiminnan varmistaminen, eli regressiotestaus, on yleisin kokeiluiden kohde. Tämä voi johtua myös näkemyseroista sen suhteen, mikä nähdään osaksi jatkuvaa kokeilemista, sillä regressiotestaus voidaan nähdä myös osana ohjelmistotestausta eikä jatkuvaa kokeilemista.

Regressiotestauksen avulla voidaan varmistaa, että järjestelmään tehtävät muutokset eivät hajota tai huononna olemassaolevaa toiminnallisuutta jollakin tavalla. Kooltaan nämä muutokset voivat vaihdella paljonkin. Joihinkin ohjelmistoihin pieniä bugikorjauksia tai vanhan toiminnallisuuden refaktorointeja saattaa tulla jopa päivittäin. Kohavi ja kumppanit (Kohavi, Deng, Frasca et al., 2013) suosittelevat useiden tällaisten muutosten keräämistä yhteen, jolloin monelle tällaiselle muutokselle riittää yksi yhteinen kokeilu. Isommille refaktoroinneille tai kokonaisten järjestelmän osien uudistuksille voidaan tehdä pelkästään niille tarkoitetut kokeilut. Tällainen kaikkien muutosten kokeileminen vaatii kuitenkin yrityksiltä merkittävää panostusta kokeilemista tukevaan infrastruktuuriin (Kohavi, Deng, Frasca et al., 2013).

Mielenkiintoinen ja harvemmin käytetty kokeilun kohde on negatiiviset kokeilut. Kohavi ja kumppanit (Kohavi, Deng, Frasca et al., 2013) esittävät, että ohjelmistoissa pitäisi aika ajoin tehdä niitä huonontavia kokeiluita. Näiden tarkoituksena on varmistaa tai todistaa vääräksi vallitsevia ohjelmistoa koskevia käsityksiä ja oletuksia. Lisäksi negatiivisten kokeiluiden tekeminen parantaa organisaation tietämystä ohjelmiston keskeisimmistä metriikoista. Tämä voi auttaa ohjelmiston kokonaisarviointikriteerien ja tulevien kokeiluiden laatimisessa. Negatiivisten kokeiluiden tarkoitus on siis huonontaa ohjelmistoa väliaikaisesti lyhyellä aikavälillä, mutta parantaa sitä pitkällä aikavälillä.

Jatkuvassa kokeilemisessa tehdään yleisesti A/B-testejä, jotka testaavat jonkin tietyn muutoksen vaikutusta käyttäjien käyttäytymiseen. Mutta entä jos halutaankin testata usean testiversion muutoksien vaikutuksia kontrolliversioon nähden ja lisäksi toisiinsa nähden? Tällöin voidaan tehdä monimuuttujatesti (eng. MultiVariable testing) (Kohavi, Longbotham et al., 2009). Monimuuttujatestin etuna on, että sen avulla voidaan testata monta eri muuttujaa lyhyessä ajassa. Jos esimerkiksi yhden kokeilun tarvitseman datan keräämisessä kestää kuukauden verran ja sovellukseen halutaan tehdä kolme erillistä kokeilua, niin niiden suorittamiseen peräkkäin kuluu kolme kuukautta. Monimuuttujatestillä voidaan kuitenkin suorittaa nämä kaikki samanaikaisesti, jolloin ne voidaan suorittaa yhdessä kuukaudessa. Monimuuttujatestien toinen etu on, että

niillä saadaan selville eri muutosten vaikutukset toisiinsa. On mahdollista, että joillain muutoksilla on erillään testattuna positiivinen vaikutus ohjelmistoon, mutta yhdessä niiden vaikutus onkin negatiivinen.

Monimuuttujatestien huono puoli on niiden myötä kokeilemiseen tuleva monimutkaisuus. Tämä monimutkaisuus ilmenee ensin testausvaiheessa, sillä kaikki eri muutokset pitää olla toteutettuna ja valmiina testattaviksi samaan aikaan. Analyysivaihe on myös tavallista monimutkaisempi. Erityisesti mikäli kokeilussa tarkastellaan useita metriikoita ja niitä kaikkia pitää vertailla sekä kontrolliversioon että keskenään, niin analyysivaihe voi muuttua hankalaksi. Kohavi ja kumppanit ovatkin sitä mieltä, että monimuuttujatestit eivät yleisesti ottaen ole tarpeellisia, sillä erillisillä muutoksilla ei yleensä ole tilastollisesti merkittävää vaikutusta toisiinsa (Kohavi, Deng, Frasca et al., 2013). Tapauksissa, joissa usealla muutoksella voidaan nähdä olevan merkittävästi vaikutusta toisiinsa on kuitenkin perusteltua käyttää monimuuttujatestausta. Samankaltaisiin tuloksiin ovat myös tulleet Tang ja kumppanit (Tang et al., 2010).

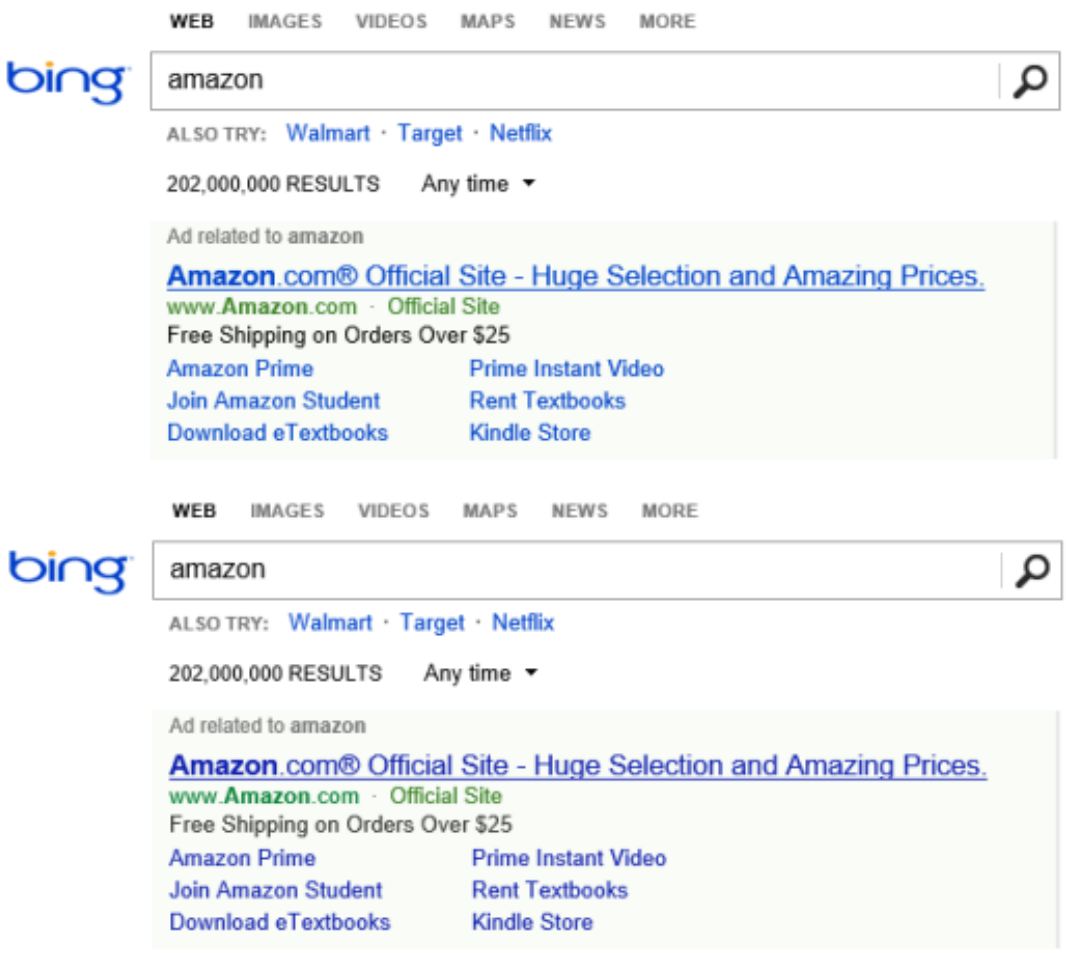
4.4.1 Kokeilujen laajuus

Kokeilujen kohteiden skaala on varsin laaja. Pienimmät kokeilut voivat esimerkiksi kohdistua jonkin käyttöliittymän napin tekstiin tai yksittäisen käyttöliittymän elementin väriin. Iso muutos taas saattaa muuttaa koko käyttöliittymän kerralla täysin erilaiseksi tai vaihtaa jonkin järjestelmän osan kokonaan uuteen. Toisaalta iso muutos voi myös olla jonkin algoritmin muutos, mikäli se on ohjelmiston toiminnan kannalta erittäin keskeisessä asemassa.

Kuva 4.9 esittää pieneen muutokseen kohdistuvan kokeilun kontrolli- ja testiversiot. Tässä kokeilussa ainoa muutos oli Bing-hakukoneen hakutulosten väreihin. Värejä vieläpä muutettiin niin vähän, että niiden eroja voi olla vaikea erottaa. Testiversion muutos oli kuitenkin merkittävä parannus kontrolliversioon nähden, sillä se paransi hakukoneen tuottoja vuositasolla miljoonilla dollareilla. Tämä esimerkki osoittaa hyvin, kuinka hyvin pienellä muutoksella voi olla iso vaikutus.

Jatkuvan kokeilemisen avulla tehtävällä ohjelmiston pienillä inkrementaalisilla parannuksilla voi myös olla huonot puolensa. Näihin törmäsi teknologiayritys Googlella työskennellyt Douglas Bowman*. Googlella jatkuva kokeileminen oli viety niin pitkälle, että selainsovelluksen linkin uudesta väristä piti kokeilla 41:tä erilaista sävyä. Joissain

*https://stopdesign.com/archive/2009/03/20/goodbye-google.html, luettu 30.4.2020



Kuva 4.9: Värien muutoksen kokeilu Bing-hakupalvelussa, muokattu (Kohavi, Deng, Longbotham et al., 2014)

tapauksissa, kuten kuvan 4.9 tapauksessa värin muutoksella voi olla isoja vaikutuksia, mutta kokeilemisessa on oltava jotain rajauksia sen suhteen, mitä on perusteltua kokeilla ja kuinka paljon.

Jatkuvaan inkrementaaliseen ohjelmiston parantamiseen keskittymisessä on riskinä, että ei tehdä enää ollenkaan isompia innovaatiota, vaan tehdään loputtomasti pieniä parannuksia*. Jatkuva kokeileminen on hyvä nähdä yhtenä työkaluna ohjelmistokehityksessä, ei ratkaisuna kaikkiin ongelmiin (Kohavi, Deng, Frasca et al., 2013). Usein hyvä lähestymistapa onkin tehdä jatkuvia pieniä inkrementaalisia parannuksia unohtamatta kuitenkaan isompien innovaatioiden roolia uusien uusien kehitysideoiden to-

*The A/B Test: Inside the Technology That's Changing the Rules of Business, https://www.wired.com/2012/04/ff-abtesting/, luettu 26.4.2020

teuttamisessa.

5 Yhteenveto

Tässä työssä tarkasteltiin jatkuvan kokeilemisen käyttöönottoa ohjelmistokehityksessä sekä useita jatkuvan kokeilemisen toteutuksen kannalta tärkeitä osa-alueita. Jatkuvan kokeilemisen käyttöönottoon liittyen käytiin läpi erilaisia liiketoiminta- ja organisaatiotason vaatimuksia sekä teknisiä vaatimuksia. Toteutuksen tärkeiksi osa-alueiksi valikoitui testi- ja kontrolliversioiden variaatioiden toteutus, datan keräys, käsittely ja analysointi sekä kokeilun kohde.

Testi- ja kontrolliversioiden variaatioiden toteutuksesta käsiteltiin useaa erilaista toteutustapaa ja niiden hyviä ja huonoja puolia. Tämän lisäksi käsiteltiin käyttäjien jakamista näihin variaatioihin ja mitä kaikkea siinä tulee ottaa huomioon. Jatkuvan kokeilemisen dataan liittyen tarkasteltiin sen keräystä, jatkokäsittelyä sekä analysointia ja mitä näiden toteutuksessa on syytä huomioida. Lisäksi käsiteltiin lyhyesti reaaliaikaisesta monitoroinnista saatavia hyötyjä jatkuvassa kokeilemisessa. Lopuksi käsiteltiin sitä, mihin kaikkeen kokeiluita voidaan kohdistaa ja miten laajoja kokonaisuuksia kannattaa kerralla kokeilla.

Työssä käytetystä lähdemateriaalista on huomioitava, että valtaosa siitä käsittelee isoissa teknologiayrityksissä tapahtuvaa jatkuvaa kokeilemista. Näissä jatkuva kokeileminen on usein hyvin laajaa ja pitkälle vietyä ja sen tukemiseksi on usein tehty laajoja jatkuvan kokeilemisen alustoja. Pienemmillä yrityksillä ei todennäköisesti ole mahdollisuutta panostaa jatkuvaan kokeilemiseen tällaisia resursseja. Työssä esitetyt jatkuvan kokeilemisen toteutustavat ja käytännöt ovat kuitenkin sovellettavissa myös tällaisille organisaatioille. Kokeileminen voidaankin ottaa aluksi käyttöön pienempänä osana ohjelmistokehitysprosessia ja laajeentaa sitä myöhemmin.

Pienemmistä yrityksistä erityisesti startup-yritykset ovat ottaneet jatkuvaa kokeilemista käyttöön onnistuneesti. On kuitenkin ohjelmistoja, joihin jatkuva kokeileminen ei sovellu, kuten ohjelmistot, jotka sisältävät liian arkaluonteista tai tietoturvaluokiteltua dataa kokeilun tekijöiden käsitelväksi.

5.1 Tarkastelu ja jatkotutkimukset

Tämä työ tarjosi kirjallisuuskatsauksen menetelmin tehdyn yleiskatsauksen jatkuvaan kokeilemiseen ja sen osa-alueisiin. Jatkuvan kokeilemisen aihepiiristä löytyy useita mielekkäitä jatkotutkimuksen kohteita. Yksi tällainen olisi empiirinen tutkimus, jossa jatkuva kokeileminen otetaan käyttöön tässä tutkielmassa käsiteltyjä menetelmiä ja käytäntöjä käyttäen. Tällaisen työn tekninen kuvaus voisi täydentää tässä työssä esiteltyjä jatkuvan kokeilemisen käytäntöjä. Joitakin tämän tyyppisiä tutkielmia onkin jo tehty kokonaisista jatkuvan kokeilemisen arkkitehtuureista. Tämä olisi erityisen kiinnostavaa pienemmässä skaalassa toteutettuna, sillä suurin osa tämän kirjallisuuskatsauksen lähdemateriaalista keskittyi isompiin organisaatioihin.

Tutkimus painottuu myös selvästi selainpohjaisiin ohjelmistoihin. Jatkuvan kokeilemisen soveltamisesta muuntyyppisiin ohjelmistoihin olisi tarpeen tehdä lisää tutkimusta. Tällaisia ohjelmistotyyppejä voisivat olla esimerkiksi sulautetut järjestelmät tai esineiden internet. Varsinkin esineiden internetin käytön sekä tutkimuksen yleistymisen myötä sen yhdistäminen jatkuvaan kokeilemiseen voisi olla varsin ajankohtainen aihe.

Jatkuvaa kokeilemista voidaan selvästi myös kehittää vielä monella tavalla. Tästä on kirjallisuudessa useita viitteitä ja muutamat tutkielmat ovatkin jo tarttuneet näihin. Yksi tällainen on jatkuvan kokeilemisen automatisoiminen käyttäen koneoppimista ja tekoälyä. Koneoppiminen ja tekoäly ovat itsessään yleisiä tutkimuskohteita tällä hetkellä, mutta niiden soveltamista jatkuvaan kokeilemiseen ei ole vielä tutkittu kovin merkittävästi.

Kirjallisuus

Auer, F. ja Felderer, M. (2018). "Current State of Research on Continuous Experimentation: A Systematic Mapping Study". *IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, s. 335–344.

Bosch, J. ja Eklund, U. (2012). "Eternal Embedded Software: Towards Innovation Experiment Systems". *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, s. 19–31.

Crook, T., Frasca, B., Kohavi, R. ja Longbotham, R. (2009). "Seven pitfalls to avoid when running controlled experiments on the web". *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, s. 1105–1114.

Deng, A. ja Shi, X. (2016). "Data-Driven Metric Development for Online Controlled Experiments: Seven Lessons Learned". *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, s. 77–86.

Elbaum, S., Rothermel, G. E. ja Penix, J. J. (2014). "Techniques for improving regression testing in continuous integration development environments". *FSE 2014: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, s. 235–245.

Fabijan, A., Dmitriev, P., McFarland, C., Vermeer, L., Olsson, H. H. ja Bosch, J. (2018). "Experimentation growth: Evolving trustworthy A/B testing capabilities in online software companies". *Journal of Software: Evolution and Process*.

Fabijan, A., Dmitriev, P., Olsson, H. H. ja Bosch, J. (2017a). "The Benefits of Controlled Experimentation at Scale". *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*.

– (2017b). "The Evolution of Continuous Experimentation in Software Product Development: From Data to a Data-Driven Organization at Scale". *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*.

Fagerholm, F., Guinea, A. S., Mäenpää, H. ja Münch, J. (2017). "The RIGHT model for Continuous Experimentation". *Journal of Systems and Software* 123, s. 292–305.

Farley, D. ja Humble, J. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional.

Giaimo, F. ja Berger, C. (2017). ”Design Criteria to Architect Continuous Experimentation for Self-Driving Vehicles”. *2017 IEEE International Conference on Software Architecture (ICSA)*.

Gupta, S., Ulanova, L., Bhardwaj, S., Dmitriev, P., Raff, P. ja Fabijan, A. (2018). ”The Anatomy of a Large-Scale Experimentation Platform”. *International Conference on Product-Focused Software Process Improvement*.

Hilton, M., Tunnell, T., Huang, K., Marinov, D. ja Dig, D. (2016). ”Usage, costs, and benefits of continuous integration in open-source projects”. *ASE 2016: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, s. 426–437.

Humble, J. (2017). ”Continuous Delivery Sounds Great, but Will It Work Here?” *ACM Queue* 15.6.

Kohavi, R., Deng, A., Frasca, B., Walker, T., Xu, Y. ja Pohlmann, N. (2013). ”Online controlled experiments at large scale”. *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, s. 1168–1176.

Kohavi, R., Deng, A., Longbotham, R. ja Xu, Y. (2014). ”Seven rules of thumb for web site experimenters”. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, s. 1857–1866.

Kohavi, R., Henne, R. M. ja Sommerfield, D. (2007). ”Practical guide to controlled experiments on the web: listen to your customers not to the hippo”. *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, s. 959–967.

Kohavi, R. ja Longbotham, R. (2017). ”Online Controlled Experiments and A/B Testing”. *Encyclopedia of Machine Learning and Data Mining*, s. 922–929.

Kohavi, R., Longbotham, R., Sommerfield, D. ja Henne, R. M. (2009). ”Controlled experiments on the web: survey and practical guide”. *Data Mining and Knowledge Discovery* 18.1, s. 140–181.

Lai, S.-T. ja Leu, F.-Y. (2015). ”Applying Continuous Integration for Reducing Web Applications Development Risks”. *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*.

Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mäntylä, M. V. ja Männistö, T. (2015). ”The highways and country roads to continuous deployment”. *IEEE Software* 32.2, s. 64–72.

Mattos, D. I., Dmitriev, P., Fabijan, A., Bosch, J. ja Olsson, H. H. (2018). "An Activity and Metric Model for Online Controlled Experiments". *International Conference on Product-Focused Software Process Improvement*, s. 182–198.

Neely, S. ja Stolt, S. (2013). "Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy)". *2013 Agile Conference*.

Pecchia, A., Cinque, M., Carrozza, G. ja Cotroneo, D. (2015). "Industry Practices and Event Logging: Assessment of a Critical Software Development Process". *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*.

Rissanen, O. ja Münch, J. (2015). "Continuous Experimentation in the B2B Domain: A Case Study". *IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, s. 12–18.

Rodden, K., Hutchinson, H. B. ja Fu, X. (2010). "Measuring the user experience on a large scale: user-centered metrics for web applications". *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, s. 2395–2398.

Ros, R. ja Runeson, P. (2018). "Continuous experimentation and A/B testing: a mapping study". *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*, s. 35–41.

Schermann, G., Cito, J. ja Leitner, P. (2018). "Continuous Experimentation: Challenges, Implementation Techniques, and Current Research". *IEEE Software* 35.2.

Schermann, G., Cito, J., Leitner, P., Zdun, U. ja C.Gall, H. (2019). "We're doing it live: A multi-method empirical study on continuous experimentation". *Information and Software Technology* 99, s. 41–57.

Shahin, M., Babar, M. A. ja Zhu, L. (2017). "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". *IEEE Access* 5, s. 3909–3943.

Tang, D., Agarwal, A., O'Brien, D. ja Meyer, M. (2010). "Overlapping experiment infrastructure: more, better, faster experimentation". *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, s. 17–26.

Tankard, C. (2016). "What the GDPR means for businesses". *Network Security* (6), s. 5–8.

Tullis, T. ja Albert, B. (2013). *Measuring the User Experience*. Morgan Kaufmann.

Vasilescu, B., Yu, Y., Wang, H., Devanbu, P. ja Filkov, V. (2015). "Quality and productivity outcomes relating to continuous integration in GitHub". *ESEC/FSE 2015:*

Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, s. 805–816.

Xu, Y., Chen, N., Fernandez, A., Sinno, O. ja Bhasi, A. (2015). "From Infrastructure to Culture: A/B Testing Challenges in Large Scale Social Networks". *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Yuan, D., Park, S. ja Zhou, Y. (2012). "Characterizing logging practices in open-source software". *Proceedings of the 34th International Conference on Software Engineering*, s. 102–112.